Master's Programme in in Mechanical Engineering

Learning Partial Differential Equations from Data

Valerii Iakovlev

Aalto University

MASTER'S
THESIS

Aalto University
MASTER'S THESIS 2020

Learning Partial Differential Equations from Data

Valerii Iakovlev

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.
Otaniemi, 15 May 2020

Supervisor: Prof. Harri Lähdesmäki
Advisor: Dr. Markus Heinonen

Aalto University
School of Science
Master's Programme in Mechanical Engineering

Author
Valerii Iakovlev

Title
Learning Partial Differential Equations from Data

School School of Engineering

Master's programme Mechanical Engineering

Supervisor Prof. Harri Lähdesmäki

Advisor Dr. Markus Heinonen

Date 15 May 2020 **Pages** 47 **Language** English

Abstract

Partial differential equations (PDEs) are ubiquitous in science and engineering for their ability to model the behavior of various systems. In science, PDEs are used to model a multitude of phenomena ranging from quantum mechanics to brain modeling. In engineering, PDEs form the basis of most simulation software which is used to model processes such as heat transfer and collapse of structures.

Many systems of interest already have accurate PDE-based models, but some systems are so complex that describing them in terms of PDEs possesses a serious challenge. This process might be simplified with the help of machine learning. When there is enough observations about a system, PDEs governing this system might be "learned".

This work proposes a method of learning black-box approximations of PDEs from data. The method is based on graph neural networks which allows it to be used on unstructured spatial grids. Furthermore, the continuous-time nature of the method makes it robust against perturbations in the time grid.

Experiments demonstrate that the method can be applied to different types of PDEs, can be used on solution domains of different shapes, supports different boundary conditions and is able to work with noisy data.

Contents

1. Introduction

Nowadays, it is hard to overestimate the importance and ubiquity of partial differential equations (PDEs) in science and engineering. In science, PDEs are used to model a multitude of phenomena ranging from quantum mechanics to brain modeling. To name a few prominent examples of PDEs, there are Maxwell's equations describing the propagation of electromagnetic fields, Navier-Stokes equations describing the behavior of viscous incompressible Newtonian fluids, Schrödinger equation describing quantum systems, Cauchy momentum equation with appropriate constitutive relations describing the behavior of elastic bodies and many more. In engineering, PDEs are the basis of most simulation software which is used to model various processes ranging from heat distribution to collapse of structures.

Numerous PDEs have already been discovered. They describe various phenomena in different fields of science and engineering and are widely used to guide everyday decisions of practitioners in these fields. Governing equations for some systems can be directly defined from conservation principles like conservation of mass, momentum, and energy. Other systems, like those encountered in e.g. finance or neuroscience, require modeling assumptions to make them amenable to being described as PDEs. Some systems are so complex that deriving their governing equations possesses a serious challenge. The process of deriving a PDE describing the behavior of a system is called the discovery of a PDE. The standard approach to discovering a PDE requires strong mathematical skills and sufficient domain knowledge which might be a limiting factor due to complexity of some systems. Today, when data is abundant, machine learning methods might help to discover governing equations directly from observation as will be discussed later in this section.

The state of most systems governed by PDEs cannot be described analytically and must be obtained by using numerical methods such as, for example, the Finite Element Method (FEM). Despite years of progress in the field of numerical

solutions of PDEs and availability of highly efficient solvers, a sufficient amount of time is still required to obtain solutions for large and complex systems. This problem is particularly noticeable if repeated numerical solutions are required which is the case in e.g. optimization and uncertainty quantification. The standard way to address this problem is to use dimensionality reduction techniques. The main idea of such techniques is to operate on a low-dimensional representation of a high-dimensional system. The state of the system is evaluated in a low-dimensional space and the low-dimensional state is then projected onto the original space to obtain the approximate solution for the high-dimensional system. Basis of the low-dimensional approximation space is obtained by repeatedly solving for the state of the full system with different configurations (parameters) and applying appropriate numerical techniques to extract the basis.

Despite being applicable to a large body of problems, dimensionality reduction techniques cannot be expected to work equally well in all cases. Standard techniques approximate the solution manifold by a subspace of the solution space. In some cases, the solution manifold cannot be accurately approximated by the subspace and dimensionality reduction techniques fail to provide accurate low-dimensional representations [1].

1.1 Recovery of Governing Equations

After outlining some of the problems that might arise when dealing with PDEs, it becomes clear that standard approaches to dealing with these problems are limited. Over the last years, many works showed that data-driven methods could be a strong competitor to standard methods.

The problem of recovering governing equations of dynamical systems from observations is not new. The area of system identification (SI) has been concerned with this problem for a long time. Its main goal is to use observations of a dynamical system to construct a mathematical model that approximates the behavior of the system. There are two types of models in SI: black-box models and grey-box models. Both types are parametric. Black-box models use no physical insight and are not interpretable. The process of selection of a black-box model consists of selecting the form of the model and then finding its parameters such that the model fits the data as accurately as possible. Grey-box models, on the other hand, are based on physical laws but contain unknown parameters that should be estimated from the data.

The main limitation of standard SI methods is the requirement to provide a parametric model that will be used to approximate the observed dynamics. This

problem led to multiple works that attempted to recover the symbolic form of the governing equations without assumptions about the model form [2, 3, 4]. These methods use genetic programming [5] to generate a set of candidate nonlinear models and select the ones which are most accurate at describing the data. A new approach called Sparse Identification of Nonlinear Dynamics (SINDy) was introduced in [6]. The method uses a dictionary of terms that might be present in the governing equations of the dynamical system of interest. It then leverages the fact that many governing equations contain a small number of such terms and uses sparse regression techniques to enforce this constraint. This produces parsimonious interpretable models that accurately describe the data. Similar dictionary-based sparse regression techniques were applied to PDEs in [7, 8] and were shown to successfully recover linear and nonlinear PDEs of various complexities.

1.2 Deep Learning Models as Surrogates of PDEs

The second problem which was outlined is the amount of time and computational resources required to obtain numerical solutions for large and complex systems. This problem is aggravated if solutions must be evaluated repeatedly which is the case in e.g. optimization and uncertainty quantification. Luckily, data-driven methods developed over the last couple of decades have been shown to be very successful in dealing with this problem.

Neural Networks (NNs) have been used for solving PDEs for a long time with first works dating back to 1997 [9] where the solution was approximated by a neural network and its parameters were found by minimizing violation of the governing equations at a set of fixed spatial locations. Two decades later, a similar method called physics informed neural networks (PINNs) was introduced in [10]. The main idea of the method is to approximate the solution by a neural network and find its parameters by minimizing a composite loss function. The loss function consists of terms related to violation of the governing equations and terms related to violation of the initial and boundary conditions. The main advantage of this method is that the model can leverage the physics of the problem which makes it possible to obtain solutions that satisfy all physical constraints that are present in the system. Shortly after the introduction of PINNs multiple similar methods appeared [11, 12, 13, 14]. These methods are not exactly identical, but all share the same idea of fitting a neural network by minimizing how much it violates the physics, boundary and initial conditions of the problem.

Neural networks have also been used for alleviating shortcomings of some

classical approaches. In [15] extended dynamic mode decomposition (EDMD) [16] was improved by incorporating a neural network into the method. EDMD requires a priori selection of the dictionary of observables, but this selection is not trivial especially in the case of nonlinear or high-dimensional systems. Instead of selecting the dictionary manually, a neural network was used to learn it. This lead to more optimal selection of the dictionary which improved the accuracy of the method. In [17] accuracy of the reduced order method (ROM) was improved by using a neural network for learning a correction term for the reduced system. This approach achieved better accuracy than the standard method, especially for nonlinear problems where the standard method is known to be inaccurate. A similar idea was used in [18] where a neural network was applied to learning a closure for the LES version of the Navier-Stokes equations. Accuracy of the learned closure was shown to be comparable with the widely used constant-coefficient and dynamic Smagorinsky models.

Another area where neural networks can be applied is obtaining surrogates of PDEs. In the case of stationary PDEs, the most common task is to evaluate the solution of a PDE as a function of a vector of parameters that might define domain shape, initial/boundary conditions, and other relevant properties. In this case, NNs are used for approximating the mapping from the vector of parameters to the corresponding solution or a function of the solution. This approach was used in [19], where a fully-connected neural network (FCNN) represented a mapping from a vector of parameters to a scalar function of the solution. Similar methods were used in [20, 21] where a convolutional neural network (CNN) was used to map an input field (e.g. conductivity field) to the corresponding solution field. One particularly interesting line of work in this area is data-free methods [14, 22] which do not require any simulation or experimental data to construct a surrogate model. This was achieved by incorporating the PDE residual into the loss function and minimizing it over the model parameters.

In the case of dynamical PDEs, the approach is typically different. Instead of trying to learn a direct mapping from the vector of parameters (which now includes time) to the solution, the goal is to find a function that approximates the dynamics or the evolution of the system. This is usually done in two ways: either learning a mapping from the current state to the next state or learning a function that approximates the temporal derivative of the state. The first approach is very common and was used in [23, 24, 22] where the goal is to learn a function that advances the current state forward by a fixed time step. Training a model with such methods requires training data to be evaluated with a fixed time step. The main drawback of the resulting models is their inability to work with time steps

that differ from the one with which they were trained. The second approach is much less common and was used in a handful of works e.g. [18, 25]. Such methods allow to train models and obtain predictions with varying timesteps. Also, the resulting models can be used with standard time-integration schemes instead of relying on ad hoc time-stepping methods.

1.3 Graph Networks for Learning Dynamical Systems

One approach to learning dynamical systems is based on the decomposition of the system into its constituent parts which are commonly referred to as objects. This approach was used in e.g [26, 27, 28] where the evolution of the system was modeled by object- and relation-centric functions. Such functions are not attempting to predict the next state of the system by looking at the whole system at once. Instead, they consider each object in the system separately and predict how it is going to change based on its state and states of the corresponding context objects. Context objects are selected in such a way that they affect the state of the current object. It also was shown that such object-centered representations help the models generalize to previously unseen scene configurations. While not using Graph Networks (GNs) explicitly, these works implement functions that realize the relational inductive bias which is one of the main features of GNs [29].

GNs were used to model physical systems in [30] where it was shown that this approach is most effective in systems where objects have common structure i.e. share the same functionality. In [31] GNs were applied to model dynamical systems governed by PDEs. At the time of writing all GN-based models applied to dynamic PDEs are discrete-time which means they learn a function that advances the state of the system forward by a fixed amount of time. Limitations of this approach were discussed in the previous subsection.

1.4 Scope of the Work

All surrogate models for dynamical PDEs available in the literature can be roughly divided into three groups: models based on fully connected neural networks, models based on convolutional neural networks and models based on graph networks. All models are limited in various ways.

FCNN-based models require explicit parametrization of the system they are trying to approximate. Such things as initial conditions, boundary conditions, domain shape and size, time step and other properties need to be provided to the model. Providing efficient parametrizations for such a large number of configurations

is a challenging task. Furthermore, a lot of data would be required to train a sufficiently accurate model. Even if it was possible to obtain such amount of data, the model would be able to operate only within the range of parameters on which it was trained.

CNN-based methods operate on the current state of the system to produce either the next state or the temporal derivative of the state. Some methods consider the whole input at once which makes them susceptible to the same problems as in the case of FCNN-based methods. Nonetheless, it is possible to sidestep this problem by forcing the model to consider only a local portion of the input and make local predictions as was done in e.g. [23]. This approach is much less sensitive to changes in initial conditions, boundary conditions and domain size as it considers only local information which makes it similar to GN-based methods. Despite all these advantages, CNN-based methods can be applied only on domains with simple shapes where convolutions can be efficiently evaluated.

Graph-based methods applied to static [32] and dynamical [31] PDEs are particularly interesting. Due to their inductive bias, they consider only local information which makes them less sensitive to changes in initial/boundary conditions and domain size. Furthermore, GNs, in contrary to CNNs, are not restricted to pixel representation of the input which makes the GN-based model independent from the domain shape. These advantages make GN-based models a good choice for developing accurate data-efficient surrogate models for dynamical PDEs.

After considering advantages and disadvantages of various approaches, the scope of this work is restricted to developing a continuous-time GN-based surrogate model for dynamical PDEs. It should be possible to train the model on data unevenly spaced in time and make predictions at arbitrary time points. Furthermore, the model should not be too sensitive to changes in initial and boundary conditions, and should easily generalize to new solutions domains.

Beyond already highlighted advantages, development of such a model would allow to extend closure learning methods as [18] to arbitrary domain shapes and meshes which could highly increase the accuracy and efficiency of PDE-based simulations.

1.5 Outline

The thesis is organized as follows: Chapter 2 covers theoretical topics that are central to this work. Topics such as machine learning, neural networks, graph networks, parameter estimation, the finite element analysis and adjoint method are covered in details sufficient to understand the content of Chapter 3 which

describes the method developed in this work. Chapter 4 contains results of various experiments conducted to demonstrate some properties of the developed method. Finally, results of the experiments are discussed and interpreted in Chapter 5.

2. Theory

This work is very interdisciplinary and uses concepts from such areas as the Finite Element Method, partial and ordinary differential equations, numerical methods, deep learning, and optimization. This chapter provides an overview of the topics which are considered central for this project.

Section 2.1 covers the basic concepts of machine learning that are relevant to this project. Section 2.2 provides an overview of neural networks and shows how they can be used in the regression setting. Section 2.3 introduces graph networks which are extensively used in this project. All data used in this project was obtained by solving partial differential equations using the Finite Element Method. This is a huge topic and it cannot be covered in one section. Nonetheless, an overview of the central ideas and methods is provided in Section 2.4. Finally, Section 2.5 covers the adjoint method which was used for evaluating the gradients of parameters of the models.

2.1 Machine Learning

Machine learning is concerned with the development of learning algorithms. A learning algorithm can be defined as a computer program that is able "to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [33]. This is a very broad definition that covers numerous tasks, performance measures and experiences. Therefore, only a small number of examples that are relevant to this work will be presented.

2.1.1 Experiences

Machine learning algorithms can be divided into two categories: supervised and unsupervised. Belonging to either of the categories defines what kind of experience the model is allowed to have. The most common way to represent experience is

through a data set $\mathcal{D}$ which contains a finite number of data points. In the case of supervised learning, each data point is a pair (X, Y) where X is called an example and Y is called the target. The goal of a supervised algorithm is to learn how targets Y depends on examples X. In the case of unsupervised learning algorithms the dataset contains only examples X. The goal of this type of algorithms is to learn useful representations of the data e.g. approximate the data generating distribution, find clusters or uncover lower-dimensional representations. This work is concerned only with supervised learning algorithms. Therefore, subsequent sections will cover only the topics that are related to this category.

2.1.2 Tasks

As was described in the previous subsection, supervised learning algorithms are allowed to have access to a dataset $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^{N}$. Definition and properties of X_i and Y_i define the task. There is a wide variety of tasks that supervised learning algorithms can do and covering all of them is not the purpose of this section. Therefore, only the two most common tasks are mentioned. The two most common tasks in supervised machine learning are regression and classification. For both tasks, the standard way of representing an example is $X_i \in \mathbb{R}^n$ where each element of X_i defines the value of the corresponding feature and n is the number of features. The definition of the target is what makes the difference between regression and classification. In regression the target is typically defined as $Y_i \in \mathbb{R}$ while in classification $Y_i \in \{1, ..., k\}$ where $k \in \mathbb{N}$ is the number of classes. Only regression tasks are considered in this project. Therefore, subsequent sections will cover only the topics that are related to this class of tasks.

2.1.3 Regression

Regression is a class of tasks that are encountered in supervised machine learning. This setting assumes the presence of a dataset $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^{N}$ where $X_i \in \mathbb{R}^n$ and $Y_i \in \mathbb{R}$. The goal of regression is to learn how examples and targets are related or, in other words, learn to predict Y_i given X_i. This task can be accomplished by asking the learning algorithm to produce a function $f : \mathbb{R}^n \to \mathbb{R}$ that relates examples and targets. The main question now is, how to find such a function? The most common approach is to define a parametric function $f(x; \theta)$ that is completely determined by its parameters $\theta \in \mathbb{R}^m$ and select the parameters in such a way that an appropriate performance measure is optimized. One such performance measure can be obtained by using the maximum likelihood estimator (MLE).

Maximum Likelihood Estimator

Let $\mathcal{D} = \{X_i\}_{i=1}^N$ consist of N independent and identically distributed (i.i.d.) data points sampled from the corresponding data-generating distribution $p_{data}(x)$. Define a parametric distribution $p_{model}(x;\theta)$. The goal of the maximum likelihood estimator is to find parameters θ such that $\mathcal{D}$ has the highest probability under the model $p_{model}(x;\theta)$. Lets define the maximum likelihood estimate of θ by θ_{ML}. Then, θ_{ML} can be found as the maximizer of the likelihood function $\mathcal{L}(\theta) = \prod_{i=1}^N p_{model}(X_i;\theta)$.

Application of the MLE to regression problems requires a few more assumptions. Let $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^N$ consist of N i.i.d. data points sampled from the corresponding data-generating distribution $p_{data}(x, y)$. Assume the following relation $Y = f(X;\theta) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is called noise and σ^2 is a fixed variance of the noise. Setting X to a specific value x gives the following per-data-point distribution $Y|x \sim \mathcal{N}(f(x;\theta), \sigma^2)$. Now, the maximum likelihood estimate can be found by maximizing the following likelihood function $\mathcal{L}(\theta) = \prod_{i=1}^N p_{model}(X_i, Y_i;\theta)$. For various reasons it is advised to not use the likelihood function directly, but rather take a natural log transform of it to obtain the log-likelihood function. The log-likelihood function is defined as

$$\begin{aligned}
\ln \mathcal{L}(\theta) &= \sum_{i=1}^N \ln p_{model}(X_i, Y_i;\theta) \\
&= \sum_{i=1}^N [\ln p_{model}(Y_i|X_i;\theta) + \ln p_{model}(X_i)] \\
&= \underbrace{\sum_{i=1}^N \ln \mathcal{N}(f(x;\theta), \sigma^2)}_{\mathcal{L}_1} + \underbrace{\sum_{i=1}^N \ln p_{model}(X_i)}_{\mathcal{L}_2} \\
&= \mathcal{L}_1 + \mathcal{L}_2.
\end{aligned}$$

It can be seen that θ affects $\ln \mathcal{L}(\theta)$ only through $\mathcal{L}_1$. Therefore, θ_{ML} can be obtained by maximizing $\mathcal{L}_1$ defined as

$$\begin{aligned}
\mathcal{L}_1 &= \sum_{i=1}^N \ln \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(Y_i - f(X_i;\theta))^2\right) \\
&= \sum_{i=1}^N \left[-\ln \sqrt{2\pi}\sigma - \frac{1}{2\sigma^2}(Y_i - f(X_i;\theta))^2\right] \\
&= \text{const} - \sum_{i=1}^N \frac{1}{2\sigma^2}(Y_i - f(X_i;\theta))^2
\end{aligned}$$

which is equivalent to minimizing the mean squared error (MSE) defined as $\frac{1}{N}\sum_{i=1}^N (Y_i - f(X_i;\theta))^2$. This suggests that the MSE is the performance measure

that can be used in a regression problem to obtain the maximum likelihood estimate of the model's parameters. It should be noted that this form of the log-likelihood function and its correspondence with the MSE appeared due to the assumption that the noise ϵ is normally distributed. Selection of a different noise model would lead to a different expression for the log-likelihood function. For example, if ϵ was assumed to have Laplace distribution, maximization of the log-likelihood function would be equivalent to minimization of the mean absolute error.

The next question is, how to minimize the performance measure? One of the most common approaches to address this problem is to use gradient-based optimization methods such as gradient descent.

Gradient Descent

Let $L : \mathbb{R}^m \to \mathbb{R}$ be a function that has to be minimized. This function is typically called the loss function. The main idea of gradient-based methods is to minimize this function over its input by starting with a random initial guess, denoted by θ_0, and repeatedly updating that guess using the following update formula $\theta^{new} = \theta^{old} - \eta \nabla L(\theta^{old})$ where $\eta \in \mathbb{R}$ is called the step size. Updates stop when the value of $\left\|\theta^{new} - \theta^{old}\right\|$ becomes sufficiently small.

This approach can be used to minimize the MSE by defining

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} (Y_i - f(X_i; \theta))^2$$

and evaluating its gradient as

$$\begin{aligned} \nabla L(\theta) &= \frac{1}{N} \sum_{i=1}^{N} \nabla (Y_i - f(X_i; \theta))^2 \\ &= -\frac{1}{N} \sum_{i=1}^{N} 2(Y_i - f(X_i; \theta)) \nabla f(X_i; \theta) \end{aligned}$$

which can be evaluated by computing $\nabla f(x; \theta)$.

Evaluation of $\nabla L(\theta)$ requires $\nabla f(X_i; \theta)$ to be computed at each datapoint. This might be too costly if N is large. This problem is typically approached by noting that $\nabla L(\theta)$ can be expressed as

$$\nabla L(\theta) = \mathbb{E}_{X_i, Y_i \sim \hat{p}_{data}} [\nabla (Y_i - f(X_i; \theta))^2],$$

where $\hat{p}_{data}$ is empirical distribution defined by the dataset. This shows that $\nabla L(\theta)$ is an expectation and the expectation can be estimated using a random subset of the available data. This motivates the use of stochastic gradient descent (SGD)

where a subsample B of size M is taken from the dataset and $\nabla L(\theta)$ is defined as

$$\nabla L(\theta) = \frac{1}{M} \sum_{(X_i, Y_i) \in B} \nabla (Y_i - f(X_i; \theta))^2.$$

2.2 Neural Networks

The specific type of neural networks used in this project is called a feedforward fully-connected neural network. In general, a feedforward neural network is a parametric model that can be defined in terms of a function composition

$$f(x) = f_N \circ f_{N-1} \cdots f_2 \circ f_1(x),$$

where each f_i is parameterized by its own set of parameters. Here N is called the depth, f_N is called the output layer and layers from 1 to $N-1$ are called hidden layers.

In the case of fully-connected neural networks, functions f_i are defined as

$$f_i(h) = \sigma\left(W_i h + b_i\right)$$

where $W_i \in \mathbb{R}^{n \times m}$ is a matrix of weights and $b \in \mathbb{R}^n$ is a vector of biases with n and m being layer-specific. Weights and biases parameterize each layer of the neural network and are learned during training. Function σ is a nonlinear function applied element-wise to its input and is commonly called the activation function. Typical choices of σ include hyperbolic tangent defined as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

and ReLU defined as

$$\text{ReLU}(x) = \max(0, x).$$

The importance of activation functions can be demonstrated by considering the case with identity activation function and $b_i = 0$ which gives $f_i(h) = W_i h$. This allows to compute $f(x)$ as

$$f(x) = W_N(W_{N-1}(\ldots(W_2(W_1 x))))$$

which can be equivalently represented as

$$f(x) = W' x.$$

This shows that even in the case of deep neural networks, i.e. when $N > 2$, a neural network without nonlinear activation functions can be represented as a simple linear regression model. The addition of nonlinear activation functions increases the complexity of the model and, under mild assumptions, allows to use neural networks as universal function approximators [34, 35, 36, 37].

Training Neural Networks

The previous section showed how parameters of a regression model can be evaluated using the maximum likelihood estimator. Neural networks fall into the same category of models. This means that their parameters can be learned by minimizing the mean squared error between the model's predictions and the targets, or, equivalently, maximizing the log-likelihood.

Evaluation of the gradient of the MSE requires computation of $\nabla f(x;\theta)$ at each datapoint. For simple models like linear regression, where $f(x;\theta) = \theta^T x$, the gradient can be evaluated in the closed-form. However, attempting to calculate the gradient of the loss function w.r.t. the neural network's parameter in the same manner is problematic. It would lead to lengthy impractical expressions that are expensive to evaluate. Luckily, analytical differentiation is not the only type of differentiation that exists. Other types such as numerical, symbolic and automatic differentiation are widely used in various fields of science and all have their advantages and disadvantages. The type that is most widely used for training neural networks is automatic differentiation.

Automatic Differentiation

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a function that needs to be differentiated w.r.t. its input x. This implies that the Jacobian of f, defined as $J_f = \frac{\partial f_i}{\partial x_j}$, has to be calculated. Automatic differentiation provides two methods for doing that: forward mode differentiation and reverse mode differentiation.

Forward mode differentiation allows to calculate one column of J_f at the cost of a single evaluation (forward pass) of f. This means that a function with n inputs would require n forward passes for evaluating the full Jacobian. This suggests that forward mode differentiation is most effective when $n \ll m$. Reverse mode differentiation works differently and allows to evaluate a single row of J_f at the cost of a single forward pass of f. This makes the approach most effective for cases when $n \gg m$.

When training a neural network, the gradient of a loss function $L(\theta)$ needs to be computed. The dimensionality of $L(\theta)$ is typically small while the dimensionality of θ is usually large. This is the case when reverse mode differentiation excels which explains the widespread use of this approach in deep learning.

2.3 Graph Networks

The framework of graph networks (GNs) was defined in [29] as a class of functions that can be applied to graph-structured data. In this framework, a graph is defined as $G = (u, V, E)$, where u is a global feature corresponding to the whole graph, V is the set of nodes and E is the set of edges. The graph consists of N_V nodes and N_E edges between the nodes. The set of nodes is defined as $V = \{v_i\}_{i=1}^{N_V}$, where v_i is a vector of features corresponding to node i. The set of edges is defined as $E = \{(e_k, s_k, r_k)\}_{k=1}^{N_V}$, where e_k is a vector of features corresponding to edge k and s_k with r_k define indices of the sender and the receiver nodes respectively. The central concept in the GN framework is the GN-block which is a graph-to-graph function. The GN-block consists of three "aggregation" and three "update" functions. When these functions are defined with the help of neural networks the resulting model is called a graph neural network (GNN) [38, 39].

In the last years, there has been a lot of work related to GNNs which resulted in various types of methods being developed. The types that are most relevant for this work are recurrent and convolutional.

The idea of recurrent graph neural networks is to update each node's hidden state, defined as $h_i,\ i = 1, \ldots, N_v$, by iteratively applying the following update rule

$$h_i^{(k)} = F(h_{\mathcal{N}(i)}^{(k-1)}, v_i, v_{\mathcal{N}(i)}, e_{(i,\, \mathcal{N}(i))}),$$

where the initial hidden state $h_i^{(0)}$ is initialized randomly, F is a parametric function, $\mathcal{N}(i)$ contains indices of neighbors of node i and $e_{(i,\, \mathcal{N}(i))}$ is the set containing features of edges directed from node i to its neighbors. The updates are continued until the hidden state of each node converged to some value. To ensure convergence, F must be a contraction mapping.

Convolutional GNNs are related to recurrent GNNs in that they also update the hidden state of each node based on its neighborhood. The difference is that instead of applying a contraction mapping until convergence, convolutional models define a fixed number of layers with different parameters and update the hidden state of each node at every layer. Convolutional models can be divided into two types: spectral-based and spatial-based.

Spectral-based methods deal with undirected graphs. Such a graph can be represent by the normalized graph Laplacian defined as

$$L = I_n - D^{-\frac{1}{2}} A D^{-\frac{1}{2}},$$

where A is the adjacency matrix defined as $A_{ij} = 1$ if nodes i and j are connected

and $A_{ij} = 0$ otherwise, D is the degree matrix defined as $D_{ii} = \sum_j A_{ij}$, and I_n is the identity matrix with n rows and columns. The Laplacian is symmetric and positive-semidefinite which means eigendecomposition can be used to factorize it as

$$L = U\Lambda U^T,$$

where columns of U are eigenvectors of L and Λ is a diagonal matrix with eigenvalues of L. The graph convolution of the input signal x with a filter g is defined as

$$x *_G g = U(U^T x \odot U^T g),$$

where x and g are real vectors of size n, the filter g is parameterized by a set of learnable parameters and $\odot$ is element-wise multiplication. Due to the dependence of the convolution on U, the learned filter can be applied only to graphs which have identical structure. Also, eigendecomposition is a very expensive operation which becomes a problem for large graphs.

Spatial-based methods define graph convolutions in a different way. These methods use the concept of message passing from the recurrent methods which propagate node information along edges. Models implementing this method consist of a fixed number of layers defined by K. They update the hidden state of each node i at every layer k using the following update formula defined in [40] as

$$h_i^{(k)} = \gamma^{(k)}(h_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \phi^{(k)}(h_i^{(k-1)}, h_j^{(k-1)}, e_{i,j})), \quad k = 1, \ldots, K,$$

where the initial hidden state of each node i, defined as $h_i^{(0)}$, is set to v_i which is the vector of features of that node, $e_{i,j}$ is the vector of features of the edge directed from node i to node j, $\square$ is a permutation invariant aggregation function (e.g. sum, mean, max), and $\gamma^{(k)}$ with $\phi^{(k)}$ are differentiable parametric functions.

Spatial-based methods are more efficient, more scalable and more flexible than spectral-based methods. They easily generalize to new graphs and can be applied to graphs with directed edges.

2.4 Time and Space Discretization

Partial differential equations (PDEs) describe the behavior of various physical systems. The most common approach to deriving a PDE for a given system is through the conservation laws such as conservation of mass, momentum, energy, etc. One of the simplest PDEs is the steady-state heat equation defined as $\nabla^2 u = 0$. This equation was obtained by applying the energy conservation principle and

using the Fourier's law to approximate the heat flow. On its own, this PDE does not have a unique solution since there are many functions which satisfy it. In order to ensure the existence of a unique solution, the PDE has to be augmented with appropriate boundary conditions. Heat equation defined on some domain $\Omega \in \mathbb{R}^n$ and augmented with appropriate boundary conditions, denoted by $g(x)$, result in the following boundary value problem

$$\begin{aligned} \nabla^2 u(x) &= 0, \quad \text{in } \Omega \\ u(x) &= g(x), \quad \text{on boundaries of } \Omega \end{aligned}$$

which is called the strong form due to the differentiability requirements on $u(x)$.

Analytic solutions of boundary value problems are typically not available and must be obtained using numerical methods. The finite element method (FEM) is commonly used to find approximate solutions of boundary value problems. The first step in applying the FEM to a BVP is to convert the BVP to its weak form. The weak form of the above boundary value problem can be derived by multiplying both sides of the PDE by a test function v and integrating the result by parts. For simplicity, let's set $g(x)$ to 0. Then, the weak form is defined as

$$\int_\Omega \nabla u \cdot \nabla v \, d\Omega = 0, \quad \forall v \in V,$$

where V is an appropriate function space. It can be shown [41, 42] that for many PDEs the weak form can be defined as

$$a(u, v) = l(v), \quad \forall v \in V,$$

where $a(u, v)$ and $l(v)$ are called bilinear and linear forms respectively. In this case, $a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, d\Omega$ and $l(v) = 0$. It can be shown that this formulation is equivalent to the original BVP. One of the advantages of the weak form is that it admits new solution techniques such as the Galerkin method. Assume that V is an infinite-dimensional inner product space and V_h is its finite-dimensional subspace. Then for any $u \in V$ the Galerkin method allows to find the best approximation of u from V_h. An approximate solution of a PDE in the weak form can be calculated by selecting a finite dimensional space V_h with basis $\{v_1, \ldots, v_n\}$ and solving the following system of equations for U:

$$KU = F,$$

where $K_{ij} = a(v_j, v_i)$ and $F_i = l(v_i)$, $i, j = 1, \ldots, n$. After solving for U, the

approximate solution $w \in V_h$ is defined as

$$w = \sum_{i=1}^{n} U_i v_i.$$

An example of applying the Galerkin method is shown in Figure 2.1 where the infinite-dimensional function $u(x)$ is approximated by a finite-dimensional function $\hat{u}(x)$ coming from the space of continuous piecewise linear functions. Approximation of $u(x)$ by a finite-dimensional function $\hat{u}(x)$ is called discretization in space.

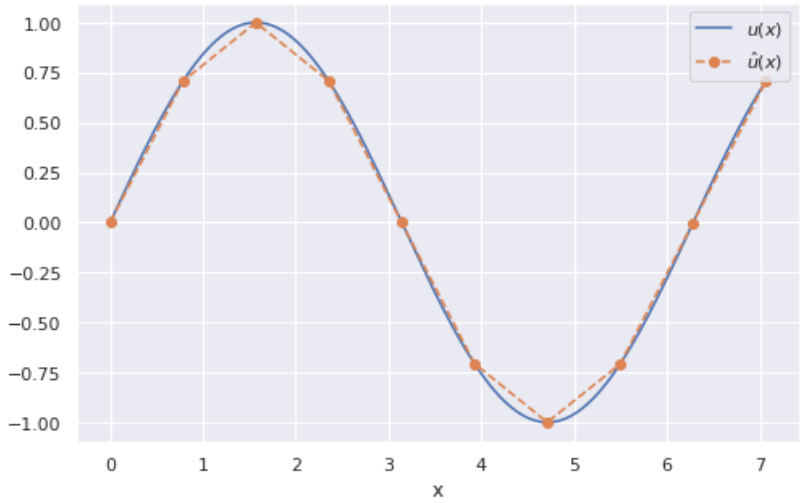

Figure 2.1. Approximation of $u(x) = \sin(x)$ by a continuous piecewise linear function $\hat{u}(x)$.

Dynamical problems, where u is a function of space and time, can be solved in a similar way but besides discretization in space, they also require discretization in time. Consider an initial boundary value problem of the form

$$\begin{aligned} \frac{\partial u(x,t)}{\partial t} &= \mathcal{L}u(x,t) + f, \quad x \text{ in } \Omega, t \in \mathbb{R}_+ \\ u(x,t) &= q(x), \quad x \text{ on } \Gamma, t \in \mathbb{R}_+ \\ u(x,t) &= g(x), \quad x \text{ in } \Omega,\ t = 0 \end{aligned}$$

where $\mathcal{L}$ is a linear differential operator, Ω is a subset of $\mathbb{R}^n$ and Γ defines its boundaries. Functions f, q and g must be sufficiently smooth.

Let's discretize this problem in time by defining u at a finite set of time points $(t_0, \ldots, t_N)$ separated by Δt and replacing the temporal derivative by a finite difference approximation. This leads to a set of N stationary problems

$$u^i - \Delta t \mathcal{L} u^i = u^{i-1} + \Delta t f, \quad i = 1, \ldots, N$$

where u^i corresponds to u at time t_i and the backward Euler scheme was used for

discretization in time. Application of the FEM starts by converting the equations above into the weak form. This is done by multiplying the equations by $v \in V$, with V being a Hilbert space, and integrating terms with second-order derivatives by parts. This leads to the following relation

$$a(u^i, v) = l(v), \quad \forall v \in V,\ i = 1, \ldots, N$$

where $a : V \times V \to \mathbb{R}$ is a V-elliptic and continuous bilinear form, and $l : V \to \mathbb{R}$ is a continuous linear form. The main problem with this formulation is that V is an infinite-dimensional space. This means that the problem cannot be solved using a computer. The idea of the finite element method is to replace V by its finite-dimensional subspace V_h and approximate the true solution by an element of V_h. Let $(\phi_1, \ldots, \phi_M)$ be the basis of V_h. Any $u_h \in V_h$ can be defined as a linear combination of the basis functions

$$u_h = \sum_{j=1}^{M} \eta_j \phi_j$$

where $\eta_j = u_h(x_j)$. Then, for every time point the infinite-dimensional problem above can be approximated by a finite-dimensional problem defined as

$$\sum_{j=1}^{M} \eta_j^i a(\phi_j, \phi_k) = l(\phi_k), \quad \forall v \in V_h,\ k = 1, \ldots, M$$

which can be equivalently written as a system of linear equations

$$A\eta = b$$

where $A_{kj} = a(\phi_j, \phi_k)$ and $b_k = l(\phi_k)$. This system can be solved for η to evaluate the approximate solution u_h.

In the case of a nonlinear problem, the resulting system of equations would be defined as

$$A(\eta)\eta = b.$$

This nonlinear system of equations can be solved using the Newton's method by defining

$$F(\eta) = A(\eta)\eta - b$$

and applying Taylor's expansion

$$F(\eta + \delta\eta) = F(\eta) + \frac{\partial F}{\partial \eta}\delta\eta.$$

Taking $\delta\eta$ such that the left hand side vanishes gives

$$\frac{\partial F}{\partial \eta}\delta\eta = -F(\eta).$$

This system of linear equations is then solved for $\delta\eta$ and η is updated as $\eta \leftarrow \eta + \delta\eta$ until convergence.

2.5 Adjoint Method

As was shown in Sections 2.1 and 2.2, parameters of neural networks are typically found using optimization methods based on gradient descent where the gradient is evaluated using automatic differentiation. In most cases this is a reasonable approach. Nonetheless, when the output of a neural network depends the solution of an ordinary differential equation, backpropagation becomes prohibitive due to large memory requirements. In this case the adjoint method [43] is a more memory-efficient alternative to backpropagation.

Consider the following problem

$$\begin{aligned} \min_{\theta} \quad & G(x,\theta) = \int_0^T g(x,t,\theta)dt \\ \text{s.t.} \quad & \frac{dx}{dt} - f(x,t;\theta) = 0 \\ & x(0) - x_0 = 0 \end{aligned}$$

where $x \in \mathbb{R}^n$ is the state of some system, g is a functional and f is a function parameterized by $\theta \in \mathbb{R}^m$ and defines how x evolves in time. The first constraint ensures that x satisfies all physical constraints of the system. The second constraint is the initial state of the system which is required to obtain a specific time-evolution of x.

One way to solve this optimization problem is to solve $\frac{dx}{dt} = f(x,t;\theta)$ for x then evaluate $G(x,\theta)$ and calculate $\nabla_\theta G(x,\theta)$. This would lead to the following expression for the gradient

$$\nabla_\theta G(x,\theta) = \int_0^T \frac{\partial g}{\partial x}\frac{\partial x}{\partial \theta} + \frac{\partial g}{\partial \theta}dt.$$

The main problem with this expression is the term $\frac{\partial x}{\partial \theta}$ which is an $n \times m$ matrix. Besides requiring evaluation of nm partial derivatives, this term is usually difficult to calculate due to complex relationship between x and θ.

Another approach that can be used to calculate $\nabla_\theta G(x,\theta)$ is called the adjoint method. It is based on the introduction of a new function $\lambda(t)$ and a few clever

manipulations that allow to calculate the gradient more efficiently.

For convenience, lets define $\frac{\partial a}{\partial b}$ as a_b and $\frac{dx}{dt}$ as $\dot{x}$. Derivation of the adjoint method starts with converting the constrained optimization problem into an unconstrained one. This is achieved by defining the Lagrangian

$$\mathcal{L} = \int_0^T \left[g(x,t,\theta) + \lambda(t)^T (\dot{x} - f(x,t;\theta)) \right] dt + \mu(x(0) - x_0),$$

where λ and μ are Lagrange multipliers.

The constraints are always satisfied since x is obtained by solving the equation in the first constraint with the initial condition defined in the second constraint. This means that $\nabla_\theta \mathcal{L} = \nabla_\theta G(x,\theta)$ and allows to set any values for λ and μ which will be used later to simplify the expression for the gradient.

The gradient can now be evaluated as

$$\nabla_\theta \mathcal{L} = \nabla_\theta G(x,\theta) = \int_0^T g_x x_\theta + g_\theta + \lambda^T \dot{x}_\theta - \lambda^T f_x x_\theta - \lambda^T f_\theta dt$$

where the term $\lambda^T \dot{x}_\theta$ can be integrated by parts as

$$\int_0^T \lambda^T \dot{x}_\theta dt = (\lambda^T x_\theta) \mid_{t=T} - (\lambda^T x_\theta) \mid_{t=0} - \int_0^T \dot{\lambda}^T x_\theta dt.$$

Substituting that into the previous equation and rearranging some terms gives

$$\nabla_\theta \mathcal{L} = \int_0^T (g_x - \lambda^T f_x - \dot{\lambda}^T) x_\theta dt + \int_0^T g_\theta - \lambda^T f_\theta dt + (\lambda^T x_\theta) \mid_{t=T} - (\lambda^T x_\theta) \mid_{t=0}.$$

This expression can be simplified by noting that $x(0)$ is a constant which makes $x_\theta(0) = 0$ and removes the last term. As was said previously, x_θ is typically difficult to calculate. Utilizing the freedom of selection of $\lambda(t)$, terms containing x_θ can be removed. Start by noting that the first term can be canceled if λ is the solution of

$$\dot{\lambda}^T = g_x - \lambda^T f_x.$$

Then, the third term can be canceled by setting $\lambda(T) = 0$ which gives the terminal condition for the equation above. These manipulations lead to the following expression for the gradient

$$\nabla_\theta \mathcal{L} = \int_0^T g_\theta - \lambda^T f_\theta dt$$

where λ is calculated by solving $\dot{\lambda}^T = g_x - \lambda^T f_x$ backwards in time with the terminal condition $\lambda(T) = 0$.

Typically, the desired state x is observed only at a finite set of time points

$(t_1, \ldots, t_{N_t})$ with corresponding observations $(x_1, \ldots, x_{N_t})$. In this case, the quantity that needs to be minimized is the error between the desired states x and the actual states $\hat{x}$ which are obtained by solving $\frac{d\hat{x}}{dt} = f(\hat{x}, t; \theta)$ with the corresponding parameters θ. The natural way to define $G(x, \theta)$ in this case is

$$G(x, \theta) = \int_0^T \sum_{i=1}^{N_t} \|x - \hat{x}\|_2^2 \delta(t - t_i) dt$$

where $\delta(t)$ is the delta function. This formulation leads to discontinuous λ. The standard way to calculate the adjoint in this case is to integrate $\dot{\lambda}$ in the intervals between the time points and add g_θ to the solution at every point t_i.

3. Method

The goal of this work is to develop a continuous-time graph-network-based surrogate model for time-dependent partial differential equations. This chapter describes the developed method and the related workflow. Section 3.1 describes how all train and test detasets were generated and what data they contain. Section 3.2 describes how the data is preprocessed and converted to graphs. Finally, Section 3.3 provides a detailed description of the approach developed in this work.

It is important to define what type of dynamical systems is considered in this work. A typical system is shown in Figure 3.1. The system is assumed to be defined on a compact subset of an n-dimensional real space. The state of the system is observed at a finite number of observation points. Locations of the observation points remain fixed.

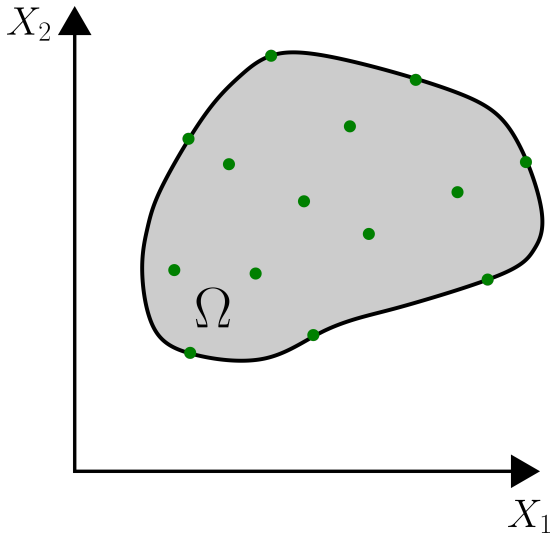

Figure 3.1. Scheme of the type of dynamical systems considered in this work. The system is defined on the domain Ω. Boundaries of the system are denoted by the solid black line. Observation points are denoted by the green dots.

3.1 Data

This section describes the general structure of data used for training and testing. Detailed explanations of the process of data generation will be provided in Chapter 4 separately for each problem.

All data used in this work is obtained by solving PDEs using the FEM. All problems were solved using FEniCS [44, 45], a popular open-source library for solving PDEs. Solutions obtained by the finite element method are available only at the nodes (Figure 3.2). Therefore, it is natural to consider nodes as observation points.

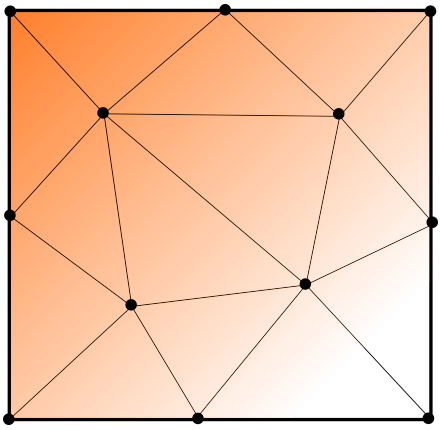

Figure 3.2. Schematic representation of a discretized domain used in FEM. Solutions provided by the FEM consist of values defined at the nodes (black dots). Values of the solution at the other parts of the domain can be obtained by interpolation.

Time-dependent problems require a time interval on which they are solved. This time interval is denoted by $[0, T]$ with T being the terminal time. After defining the time interval, M time points denoted by $(t_1, \ldots, t_M)$ are selected from that interval. The points are sorted in the ascending order with $t_1 = 0$ and $t_M = T$. Solving the problem at a time point t_i produces the corresponding solution denoted by $u^i = (u^i_1, \ldots, u^i_N)$, where u^i_j denotes the value of the solution at node j, and N is the total number of nodes. Calculating u^i for every time point t_i results in a set of solutions $U = \{u^i\}_{i=1}^M$.

In addition to U, the model needs the time points $(t_1, \ldots, t_M)$ and coordinates of the observation points $(x_1, \ldots, x_N)$, where x_i defines the position of observation point i. These three objects contain all required information from one simulation. Multiple simulations with different time points and observation points can be conducted and all data obtained from these simulations can be used for training.

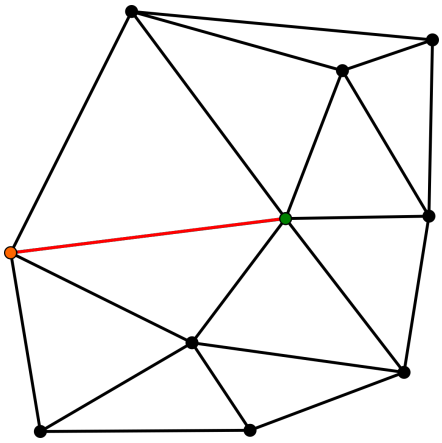

Figure 3.3. An example of the Delaunay triangulation for a set of points. The orange and green points lie on the same edge of at least one triangle. Therefore, they are considered to be neighbors.

3.2 Data Preprocessing

As was described in the previous section, the only data that is obtained from the data generation step is time points $(t_1, \ldots, t_M)$, coordinates of observation points $(x_1, \ldots, x_N)$ and a set of solutions $U = \{u^i\}_{i=1}^M$. This data needs to be converted into a form that can be used by the model.

The model is graph-based which means the data should be represented in the form of a graph. The first step in creating a graph is to define its node coordinates. It is natural to use locations of the observation points as the node coordinates. The next step is to determine which nodes are connected. Let's assume that neighboring nodes are connected. There are many ways to determine the neighbors of each node. The method used in this work is called Delaunay triangulation. It works by creating a triangulation for a set of points on a plane. As shown in Figure 3.3, if two nodes lie on the same edge of a triangle, they are considered to be neighbors. Delaunay triangulation has some attractive properties like maximizing the minimum angle within each triangle in the triangulation and containing the nearest neighbor of each node.

After defining the structure of the graph, its node and edge attributes can be filled with any values. The specific values that are used in this work will be described in the following sections.

3.3 Method Description

This section starts with the description of how to discretize a PDE and turn it into an equivalent system of ODEs using the method of lines. The right hand side (RHS) of this system is not known and should be learned from the available data.

Section 3.3.2 describes why graph-based methods are suitable for this purpose and Section 3.3.3 shows how a type of graph networks called message-passing neural networks (MPNNs) can be used to evaluate the RHS.

3.3.1 The Method of Lines

Consider a dynamical system described at the beginning of this chapter. The system is continuous in space and time. Its behavior is assumed to be governed by a PDE that can be written in a general form as

$$\frac{\partial u(x,t)}{\partial t} = F(u, \frac{\partial u}{\partial x_1}, \dots, \frac{\partial u}{\partial x_n}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \dots, \frac{\partial^2 u}{\partial x_1 \partial x_n}; , \dots), \tag{3.1}$$

where u is the state of the system, x is a vector of n spatial coordinates and t is time. When supplied with appropriate initial and boundary conditions, this equation can be solved using the method of lines (MOL) [46]. This method works by selecting N spatial points (nodes) in the system and then discretizing spatial derivatives of the PDE at these points. This gives a system of N ordinary differential equations. Solution of this system approximates the solution of the original PDE. Applying the MOL to Equation 3.1 gives the following system of ODEs

$$\begin{pmatrix} \frac{du_1(t)}{dt} \\ \vdots \\ \frac{du_N(t)}{dt} \end{pmatrix} = \begin{pmatrix} \hat{F}(x_1, x_{\mathcal{N}(1)}, u_1, u_{\mathcal{N}(1)}) \\ \vdots \\ \hat{F}(x_N, x_{\mathcal{N}(N)}, u_N, u_{\mathcal{N}(N)}) \end{pmatrix}, \tag{3.2}$$

where $\hat{F}(x_j, x_{\mathcal{N}(j)}, u_j, u_{\mathcal{N}(j)})$ defines discretization of F at the node j, $\mathcal{N}(j)$ contains indices of other nodes that are required to compute the discretization, and $x_{\mathcal{N}(j)}$ with $u_{\mathcal{N}(j)}$ are sets of positions and function values corresponding to nodes $\mathcal{N}(j)$.

The goal of this work is to learn $\hat{F}$ from data. The first thing that can be noted is the explicit dependence of $\hat{F}$ on the spatial coordinates. This is undesirable since it will cause overfitting to the particular node positions i.e. the model will fail to generalize to systems with different observation points. This problem can be solved be redefining $\hat{F}$ as

$$\hat{F}(x_{\mathcal{N}(i)} - x_i, u_i, u_{\mathcal{N}(i)}), \tag{3.3}$$

where it was assumed that F does not depend on x.

This version of $\hat{F}$ still has all the spatial information about the node positions from x_i and $x_{\mathcal{N}(i)}$, but now instead of depending on the absolute values of the positions it depends on the relative positions of the nodes.

Finally, $\hat{F}$ is parameterized by learnable parameters θ which gives

$$\hat{F}_\theta(x_{\mathcal{N}(i)} - x_i, u_i, u_{\mathcal{N}(i)}). \tag{3.4}$$

Equation 3.2 can now be redefined using $\hat{F}_\theta$ as

$$\begin{pmatrix} \frac{du_1(t)}{dt} \\ \vdots \\ \frac{du_N(t)}{dt} \end{pmatrix} = \begin{pmatrix} \hat{F}_\theta(x_{\mathcal{N}(1)} - x_1, u_1, u_{\mathcal{N}(1)}) \\ \vdots \\ \hat{F}_\theta(x_{\mathcal{N}(N)} - x_N, u_N, u_{\mathcal{N}(N)}) \end{pmatrix}. \tag{3.5}$$

Usually, the only data that is available about a dynamical system is a set of the system's states at various spatial locations and time points. As in the previous sections, let's denote the time points as $(t_1, \ldots, t_M)$, the spatial locations as $(x_1, \ldots, x_N)$ and the corresponding states as $\{u^i\}_{i=1}^M$. Let's denote the solution of Equation 3.5 by $\hat{u}$ and the value of $\hat{u}$ at a time point t_i by $\hat{u}^i$. Then, parameters θ could be found by minimizing the discrepancy between u^i and $\hat{u}^i$ at each time point t_i. The discrepancy is defined as the mean squared error (MSE) between $\hat{u}^i$ and u^i over all the time points:

$$L(\theta) = \frac{1}{M} \sum_{i=1}^{M} \left\| \hat{u}^i - u^i \right\|_2^2, \tag{3.6}$$

which should be minimized w.r.t. θ. Minimization is typically done using gradient-based optimization methods. The straightforward way of calculating the gradient of $L(\theta)$ is to use automatic differentiation, namely backpropagation, through all operations that were performed to evaluate $\hat{u}$. The main problem with this approach is that the number of operations is typically large and the amount of memory required to evaluate the gradient becomes prohibitive. Another approach proposed in [47] is based on the Adjoint Method described in Section 2.5. Since this approach does not store the solution it is memory efficient. Nonetheless, it might be unstable in some cases.

3.3.2 Motivation for GNNs

The previous subsection described the method of learning of time-dependent PDEs from data. However, particular requirements on $\hat{F}_\theta$ were not discussed. This subsection covers all the requirements and motivates the use of GNNs for representing $\hat{F}_\theta$.

From Equation 3.5 it can be noted that the sizes of $x_{\mathcal{N}(i)}$ and $u_{\mathcal{N}(i)}$ are not known and are not restricted. This means that the function $\hat{F}_\theta$ must be able to work with variable number of inputs. Furthermore, it is reasonable to make the function

independent of the order in which nodes $\mathcal{N}(i)$ are considered. Message passing neural networks satisfy all the requirements posed above and for this and other reasons, mentioned in Chapter 2, will be used in this work for evaluating $\hat{F}_\theta$.

The last thing that needs to be clarified is how to define $\mathcal{N}(i)$ for a node i. Considering the local nature of PDEs it is reasonable to assume that the state of a node i primarily depends on the states of its neighbors. Therefore, $\mathcal{N}(i)$ is defined as the set of indices of the neighbors of the node i where the neighbors are defined as described in Section 3.2.

3.3.3 Using MPNNs to Evaluate $\hat{F}_\theta$

The previous subsection motivated the use of MPNNs. This subsection describes how $\hat{F}_\theta$ can be evaluated using the MPNN framework.

In the MPNN framework, models consist of $K \geq 1$ graph layers. As was shown in Chapter 2, the state of a node j at a layer k, defined as $h_j^{(k)}$, is evaluated according to the following formula

$$h_j^{(k)} = \gamma^{(k)}(h_j^{(k-1)}, \square_{m \in \mathcal{N}(j)} \phi^{(k)}(h_j^{(k-1)}, h_m^{(k-1)}, e_{j,m})), \quad k = 1, \ldots, K, \tag{3.7}$$

where $e_{j,m}$ is a vector of features of the edge directed from the node j to the node m, $\square$ is a permutation invariant aggregation function (e.g. sum, mean, max), and $\gamma^{(k)}$ with $\phi^{(k)}$ are differentiable parametric functions.

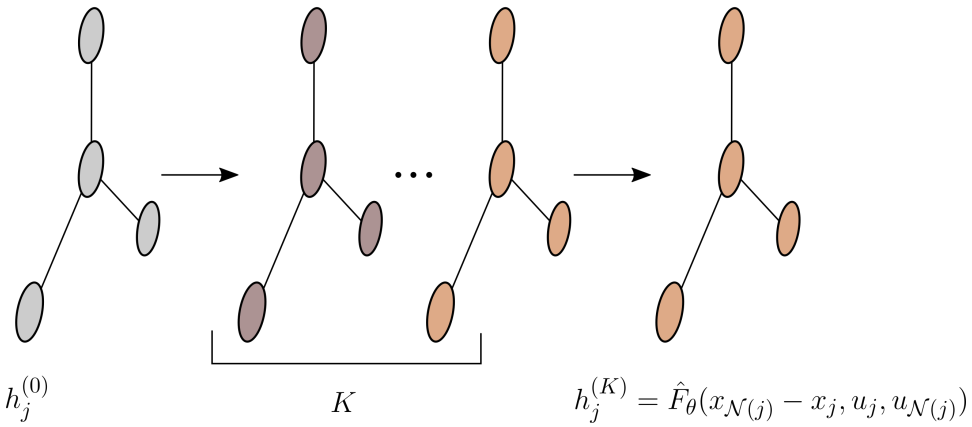

Figure 3.4. Scheme of a MPNN with K layers. Initial states of the graph nodes (grey color) are updated using Equation 3.7 at each layer. The final states of the graph nodes (tumbleweed color) are used to evaluate $\hat{F}_\theta$.

Let's say the goal is to evaluate $\hat{F}_\theta$ at time t_i. Figure 3.4 shows how it can be done. The process starts with defining the initial state of each node in the graph. The initial state of node j is defined as $h_j^{(0)} = u_j^i$, where u_j^i is the state of the system at time t_i and observation point j (since locations of the graph nodes and observation points are the same). Edge attributes are defined as $e_{j,m} = x_m - x_j$ and are kept

constant for all layers. The graph with initial states is passed through K graph layers where the state of each node is updated according to Equation 3.7. After passing through all K layers, the state of the graph at node j, defined as $h_j^{(K)}$, is assumed to give the value of $\hat{F}_\theta$ at that node. Parameters of the functions $\gamma^{(k)}$ and $\phi^{(k)}$ from graph layer $k \in \{1, \ldots, K\}$ are collectively represented by θ.

4. Experiments

4.1 Data Generation

All experimental data was generated by numerically solving partial differential equations using the finite element method. In each case, a sufficiently fine computational grid and time step were selected to obtain accurate solutions. Due to a large number of nodes in the computational and time grids, the data is typically downsampled before being used in experiments.

The computational grid is downsampled by taking all of its nodes and randomly selecting the required number of the nodes as shown in Figure 4.1. Downsampling of the fine time grid involves three steps as shown in Figure 4.2. First, a coarse time grid with a fixed time step is generated. Second, random noise is added to each time point in the coarse time grid. Time points in the coarse time grid are now random but they do not coincide with any points in the fine time grid which means there are no corresponding solutions for these time points. This problem is solved by taking each point in the coarse time grid and linearly interpolating the solution from the two closest points in the fine time grid.

In all cases, the data is generated in a similar way but with different parameters. It is convenient to define all parameters involved in the data generating process and describe the data using these parameters. Let's define the number of nodes in fine and downsampled computational grids by n and n_d respectively. All simulations are run on the time interval $[0, T]$, where T is the terminal time. The time intervals might differ for train and test data. Time steps on the fine and coarse time grids are defined by Δt and Δt_c respectively. The random noise is defined by a random variable ϵ following some probability distribution.

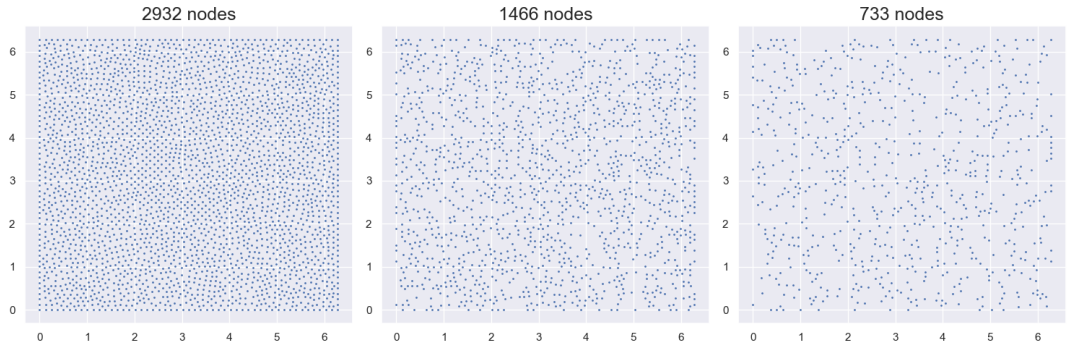

Figure 4.1. Examples of a fine computational grid with 2932 nodes and downsampled computational grids with 1466 and 733 nodes.

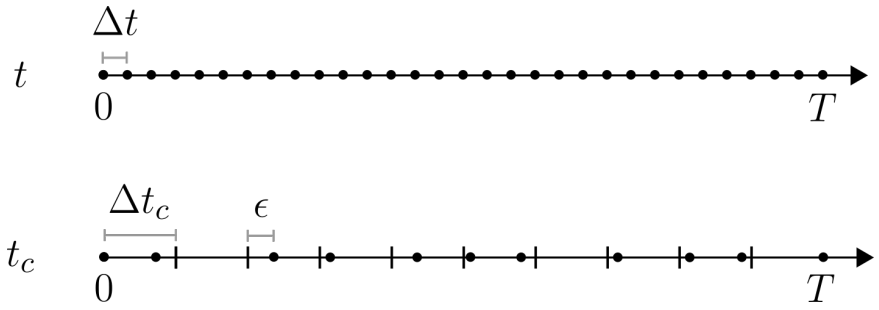

Figure 4.2. Downsampling process of a fine time grid t. First, a coarse time grid t_c with a fixed time step Δt_c is generated (corresponding time points are denoted by vertical bars). Then, random noise ϵ is added to these time points. This gives random time points denoted by black dots. The final step is linearly interpolating the solution from the two closest time points in the fine time grid (this step is not shown).

4.2 Model

The model used in this chapter is a standard message-passing neural network with the update function defined as

$$h_j^{(k)} = \gamma^{(k)}(h_j^{(k-1)}, \square_{m \in \mathcal{N}(j)} \phi^{(k)}(h_j^{(k-1)}, h_m^{(k-1)}, e_{j,m})), \quad k = 1, \ldots, K. \tag{4.1}$$

As was shown in Section 3.3.3, MPNN-based models can be defined by selecting the aggregation function, the number of graph layers K and functions $\gamma^{(k)}$ and $\phi^{(k)}$ at each graph layer $k \in \{1, \ldots, K\}$. The mean was selected to be the aggregation function i.e.

$$\square_{m \in \mathcal{N}(j)}(\cdot) = \frac{1}{|\mathcal{N}(j)|} \sum_{m \in \mathcal{N}(j)} (\cdot), \tag{4.2}$$

where $|\mathcal{N}(j)|$ is the number of neighbors of node j. The number of graph layers and functions $\gamma^{(k)}$ and $\phi^{(k)}$ might be not be the same for all experiments and for this reason are defined separately for each case.

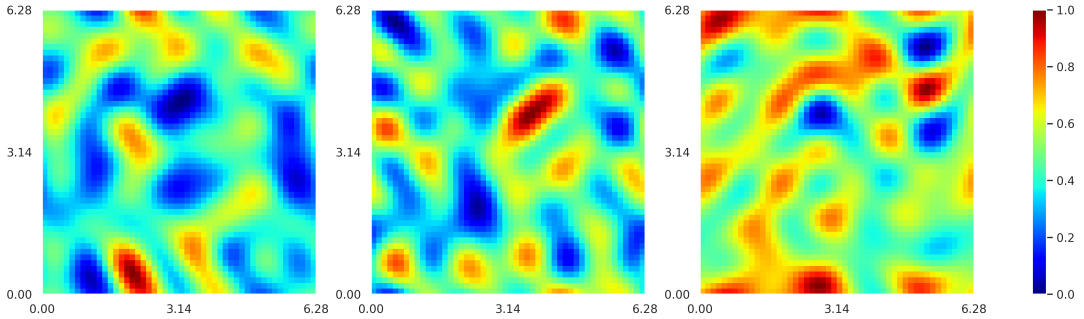

Figure 4.3. Examples of initial conditions for the convection-diffusion equation generated on a uniform 50×50 grid.

4.3 Convection-Diffusion Equation

In this section all solution domains are represented by a square Ω with sides equal to 2π and periodic boundary conditions. Therefore, the following initial-boundary value problem (IBVP) is considered

$$\begin{aligned} \frac{\partial c(x,y,t)}{\partial t} &= D\nabla^2 c(x,y,t) - v \cdot \nabla c(x,y,t), \quad (x,y) \in \Omega,\ t \geq 0, \\ c(x,0,t) &= c(x,2\pi,t), \quad t \geq 0, \\ c(0,y,t) &= c(2\pi,y,t), \quad t \geq 0, \\ c(x,y,0) &= c_0(x,y), \quad (x,y) \in \Omega,\ t = 0. \end{aligned} \tag{4.3}$$

The diffusion coefficient D was set to 0.25 and the velocity field v was set to $(5.0, 2.0)$. Here c is the concentration field. The initial conditions $c_0(x,y)$ were generated as follows:

$$c_0^{'}(x,y) = \sum_{k,l=-N}^{N} \lambda_{kl} \cos(kx + ly) + \gamma_{kl} \sin(kx + ly), \tag{4.4}$$

$$c_0(x,y) = \frac{c_0^{'}(x,y) - \min c_0^{'}(x,y)}{\max c_0^{'}(x,y) - \min c_0^{'}(x,y)}, \tag{4.5}$$

where $N = 4$ and $\lambda_{kl}, \gamma_{kl} \sim \mathcal{N}(0,1)$. Examples of initial conditions generated using this approach are shown in Figure 4.3.

Simulation data used for all experiments in this section was generated with the following parameters: $\Delta t = 0.0002$ sec, $n = 4108$, $T = 0.2$ sec for training data and $T = 0.6$ sec for testing data. In all cases the number simulations in the train and test data was set to 24 and 50 respectively.

The model used for all experiments in this section has a single graph layer with functions $\phi^{(0)}$ and $\gamma^{(0)}$ represented by multilayer perceptrons (MLPs). Let's define a MLP with Tanh activation functions, 3 hidden layers and inputs dimension a, output dimension c and hidden dimension b by $\text{NN}(a,b,c)$. Then $\phi^{(0)} = \text{NN}(4,60,40)$

and $\gamma^{(0)} = \text{NN}(41, 60, 1)$ which gives approximately 20000 trainable parameters.

In order to solve the system of ODEs defined by Equation 3.5 torchdiffeq [47] Python package was used. All models were trained on a single NVIDIA Quadro P5000 GPU for 24 hours. Training hyperparameters are listed in Table 4.1.

Table 4.1. Training hyperparameters.

ODE Solver	Adaptive Grid Adams-Bashforth-Moulton
rtol/atol	1.0e-7/1.0e-7
optimizer	Rprop [48]
learning rate	1.0e-6
loss function	Mean Squared Error
batch size	24

4.3.1 Variable Number of Nodes

The goal of this experiment is to show how the number of nodes in the spatial grid affects the performance of the models. Four downsampled spatial grids with 2991, 2244, 1497 and 749 nodes were used (Figure 4.4). The time step Δt_c used for training and testing was set to 0.01 sec which gives a time grid with 21 time point. No noise was added to the time points.

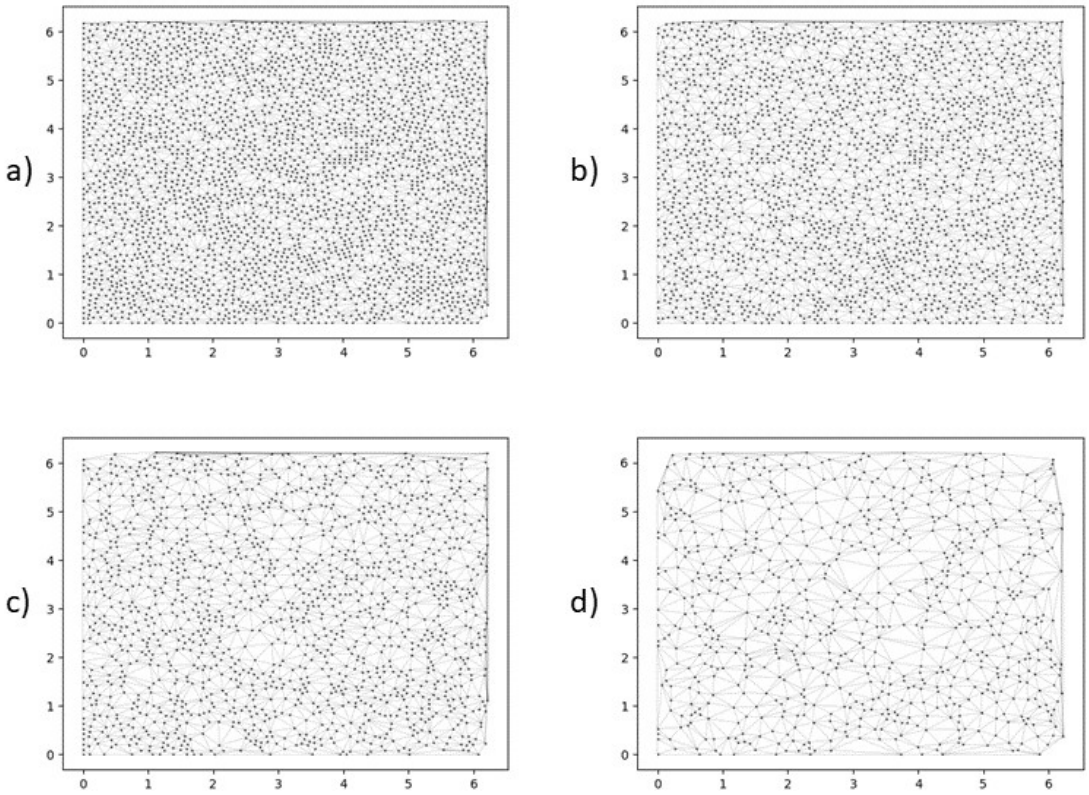

Figure 4.4. Examples of grids used in the experiment with the convection-diffusion equation. a) 2991 nodes, b) 2244 nodes, c) 1497 nodes and d) 749 nodes.

Let's define the true state of the system at time t_i as $u^i = (u_1^i, \ldots, u_N^i)$ and the predicted state of the system at the same time as $\hat{u}^i = (\hat{u}_1^i, \ldots, \hat{u}_N^i)$, where N is the

number of nodes in the spatial grid. Then, at any time point t_i the relative error between u^i and $\hat{u}^i$ is defined as

$$Err = \frac{\left\|u^i - \hat{u}^i\right\|}{\|u^i\|}. \tag{4.6}$$

The relative error will be used as the performance measure in all following experiments.

Performance of the trained models was evaluated on the test data and is shown in Figure 4.5. A comparison of predictions of the models and the true states of the system for a random test case is shown in Figure 4.6.

It can be seen that the model's performance increases with the number of nodes in the training grid. Using the same number of nodes as was used for generating the data would, apparently, lead to the smallest error. Despite relatively high errors on the coarsest grid with 748 nodes, Figure 4.6 shows that even for this grid the model is able to predict large scale features of the flow reasonably well.

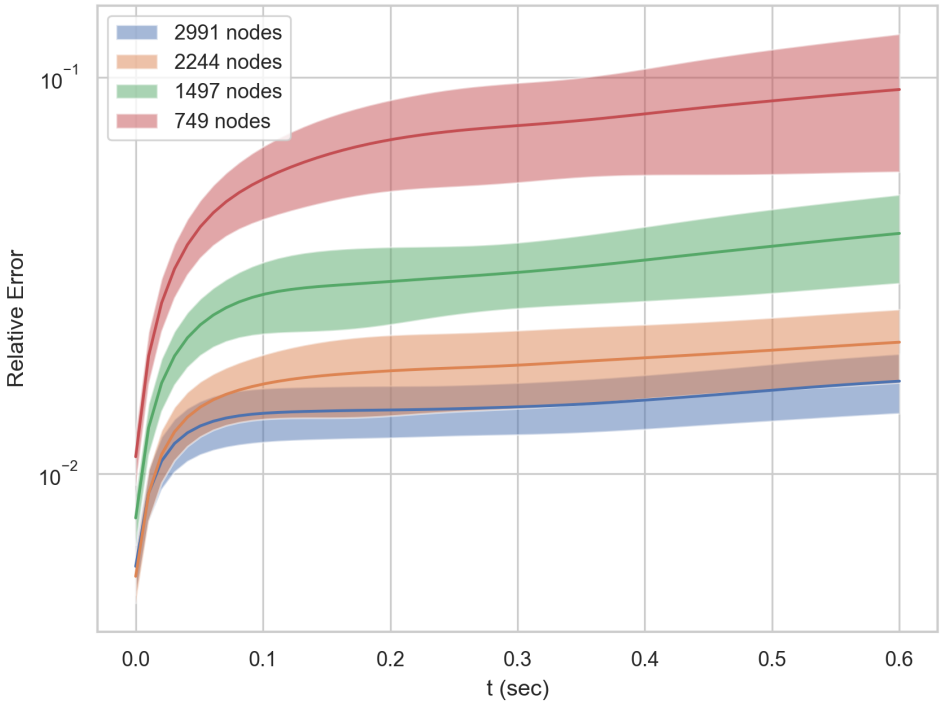

Figure 4.5. Relative test errors of models trained with different grid sizes.

4.3.2 Variable Timestep Size

The goal of this experiment is to show how the number of time points in the training data affects the performance of the models. Four time grids with 21, 6, 4 and 2 time points were used for training. The time steps Δt_c for these grids are

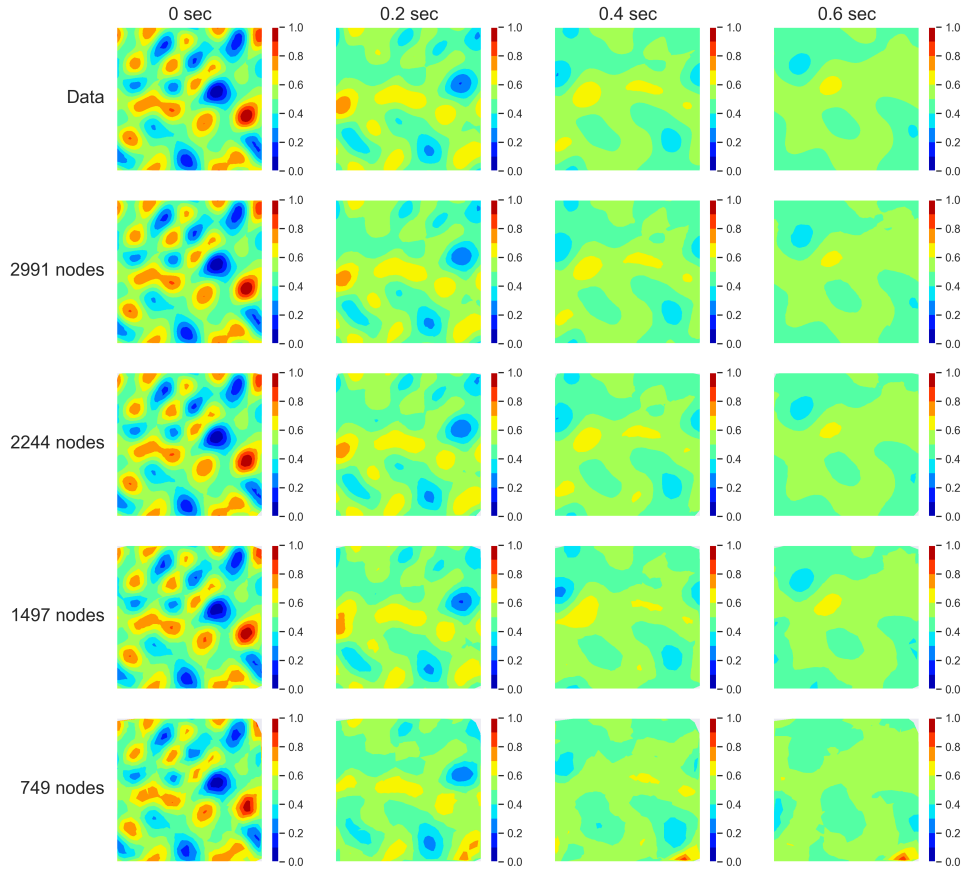

Figure 4.6. A comparison of the true states of the system with predictions of models trained on grids of different sizes. The first row shows the data, all following rows show predictions of models trained on different grids.

0.01, 0.02, 0.0667 and 0.2 sec respectively. No noise was added to the time points. The number of nodes in the spatial grid was set to 2991. All models were evaluated on the test data with $\Delta t_c = 0.01$ sec.

As in the previous experiment, the relative error between the predicted and the true state of the system is used to assess the performance of the models. Performance of the models evaluated on the test data is shown in Figure 4.7. A comparison of predictions of the models and the true states of the system for a random test case is shown in Figure 4.8. For reference, Figure 4.9 show an example of the training data with a time grid consisting of 4 time points.

Figure 4.7 shows that the model is able to recover the system's dynamics using only four time points per simulation. Increasing the number of time points does not improve the model's performance. It could be explained by the existence of the maximum time step required to capture all relevant dynamical processes in the system. In this case, the time step of 0.0667sec is below this maximum, but the

time step of 0.2sec is clearly above it. Some differences in the test relative errors of the models trained on 21, 6 and 4 time points can be noted. These differences are negligible and could be explained by better initialization of some models.

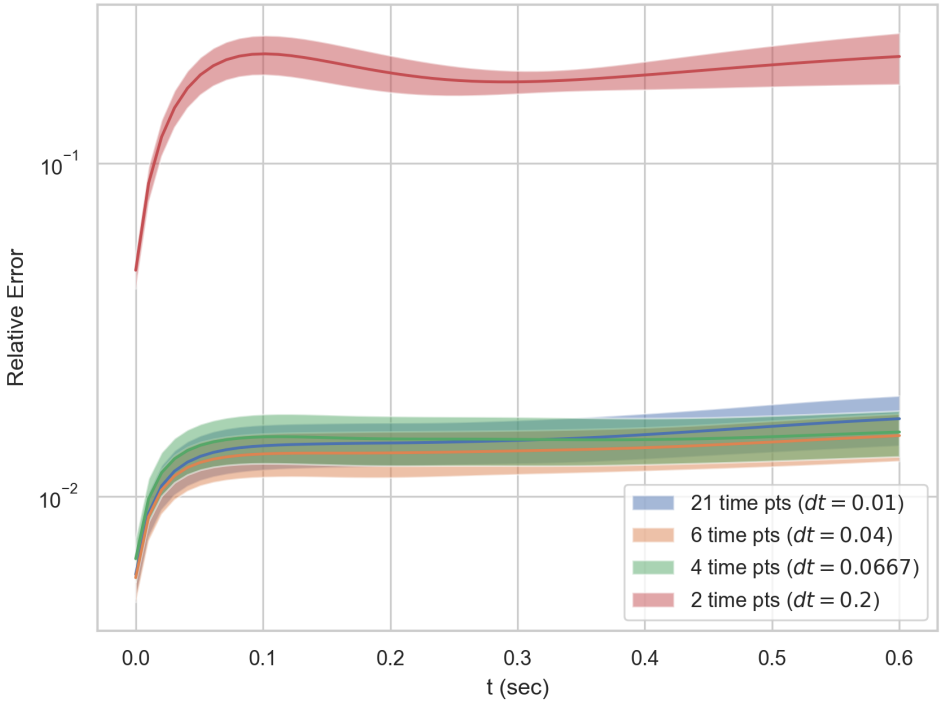

Figure 4.7. Relative test errors of models trained on different time grids.

4.3.3 Irregular Timesteps

The goal of this experiment is to show how well the model performs on the data where the system's state is observed at random time points. A time grid with $\Delta t_c = 0.02\ sec$, which gives 11 time points for training, was used. Noise applied to the time points was defined as $\epsilon \sim \mathcal{N}(0, (\frac{\Delta t_c}{6})^2)$. The number of nodes in the spatial grid was set to 2991.

Two models were considered. The first model was trained and tested on the time grid with the constant timestep Δt_c while the second model was trained and tested on the same time grid but with the noise applied to it.

As in the previous experiment, the relative error between the predicted and the true states of the system is used to assess the performance of the model. Performance of both models evaluated on their test data is shown in Figure 4.10. The figure shows that both models have similar performance. This means that the model can deal with irregular time grids without a decrease in performance. The model trained on the irregular grid performs slightly better either due to a better initialization or a better placement of the time points.

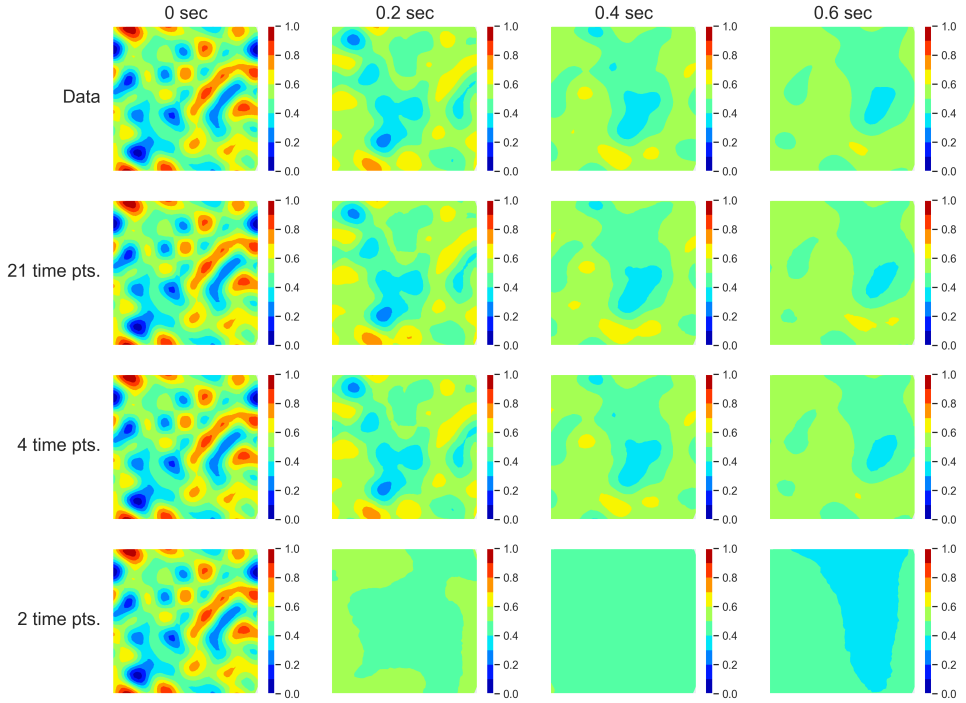

Figure 4.8. A comparison of the true states of the system with predictions of models trained on time grids of different sizes. The first row shows the data, all following rows show predictions of models trained on different grids.

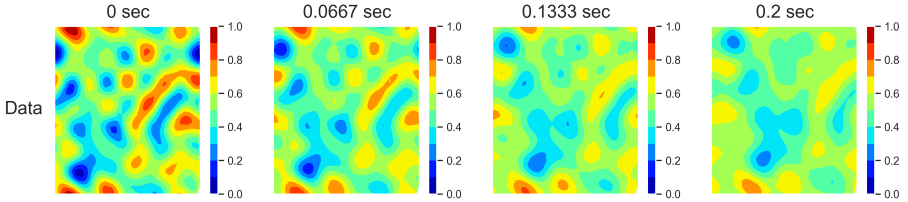

Figure 4.9. Example of a train case with 4 time points.

4.3.4 Noisy Observations

In all previous experiments, the observations were clean i.e. did not contain any noise. In this experiment, the data is corrupted with three different levels of noise by adding $\epsilon \sim \mathcal{N}(0, \sigma^2)$ to all observations. The standard deviation σ was set to $0.01,\ 0.02$ and 0.04. The largest and smallest values of the noiseless observations are 1 and 0 respectively. The number of nodes in the spatial grid was set to 2991. The time step Δt_c was set to 0.01 sec which gives a time grid with 21 time point. All models were trained on the same training data but with different levels of noise applied to it and tested on noiseless test data. Examples of an observation with different levels of noise is shown in Figure 4.11.

As in the previous experiments, the relative error between the predicted and

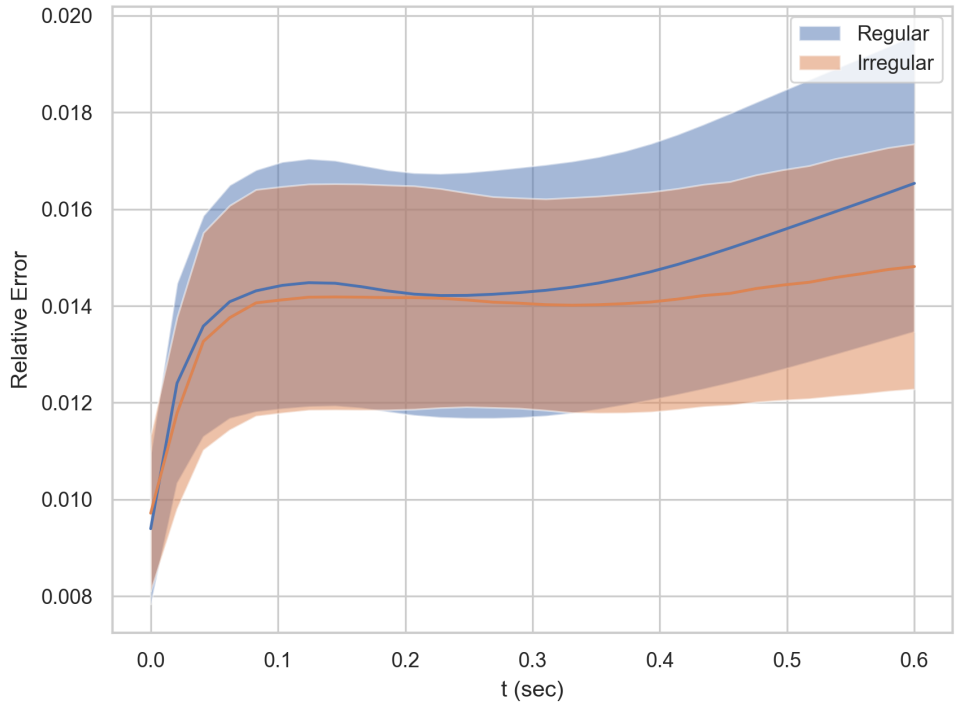

Figure 4.10. Relative test errors of models trained on a regular and irregular time grids.

the true states of the system is used to assess the performance of the models. Performance of the models evaluated on their test data is shown in Figure 4.12. A comparison of predictions of the models and the true states of the system for a random test case is shown in Figure 4.13.

The results show that the model's performance decreases as σ grows. Nonetheless, even with σ set to 0.04, the model is still able to learn a reasonably accurate approximation of the dynamics of the system. This demonstrates the model's robustness to noise.

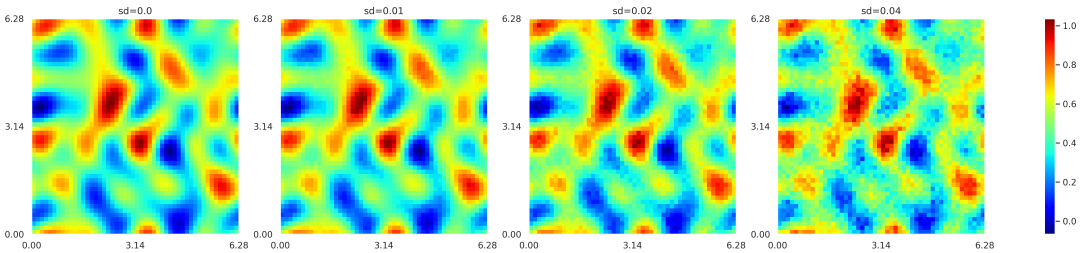

Figure 4.11. A system state with different levels of noise. Values of σ from left to right: $0.00, 0.01$, 0.02 and 0.04.

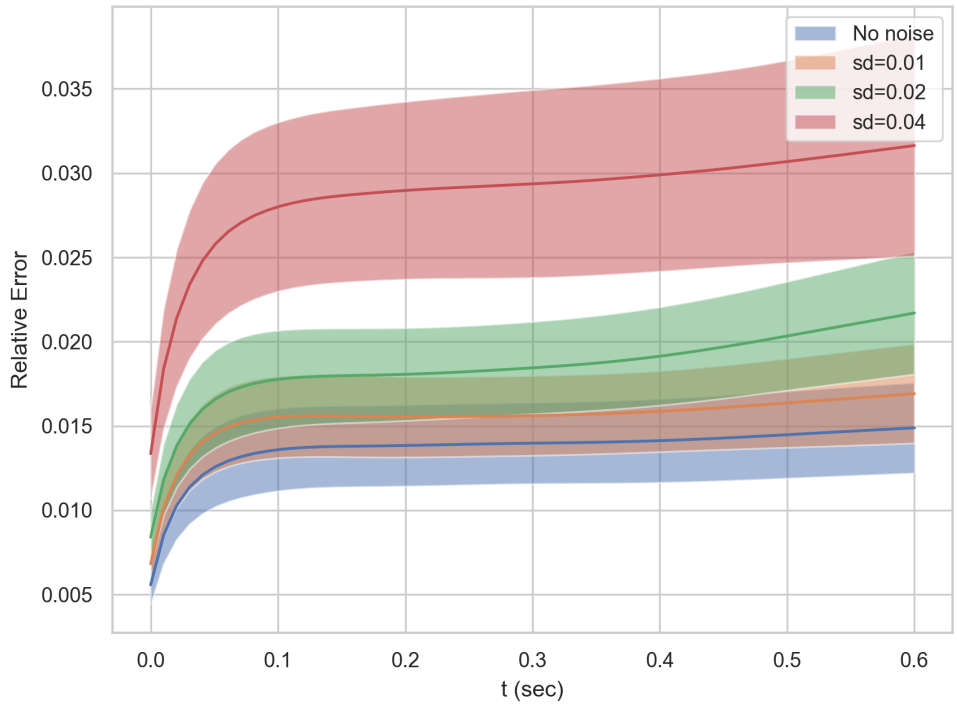

Figure 4.12. Relative test errors of models trained on observations with different levels of σ denoted by sd.

4.4 Burgers' Equations

In this section, all solution domains are represented by a square Ω with sides equal to 2π and periodic boundary conditions. Therefore, the following initial-boundary value problem (IBVP) is considered

$$\begin{aligned} \frac{\partial u(x,y,t)}{\partial t} = D\nabla^2 u(x,y,t) - u(x,y,t)\cdot\nabla u(x,y,t), \quad (x,y)\in\Omega,\ t\geq 0, \\ u(x,0,t) = u(x,2\pi,t), \quad t\geq 0, \\ u(0,y,t) = u(2\pi,y,t), \quad t\geq 0, \\ u(x,y,0) = u_0(x,y), \quad (x,y)\in\Omega,\ t=0. \end{aligned} \tag{4.7}$$

The diffusion coefficient D was set to 0.15. Here u is a velocity field with two components. The initial conditions $u_0(x,y)$ for each component were generated as follows:

$$u_0'(x,y) = \sum_{k,l=-N}^{N} \lambda_{kl}\cos(kx+ly) + \gamma_{kl}\sin(kx+ly), \tag{4.8}$$

$$u_0(x,y) = 6\times\left(\frac{u_0'(x,y)-\min u_0'(x,y)}{\max u_0'(x,y)-\min u_0'(x,y)} - 0.5\right), \tag{4.9}$$

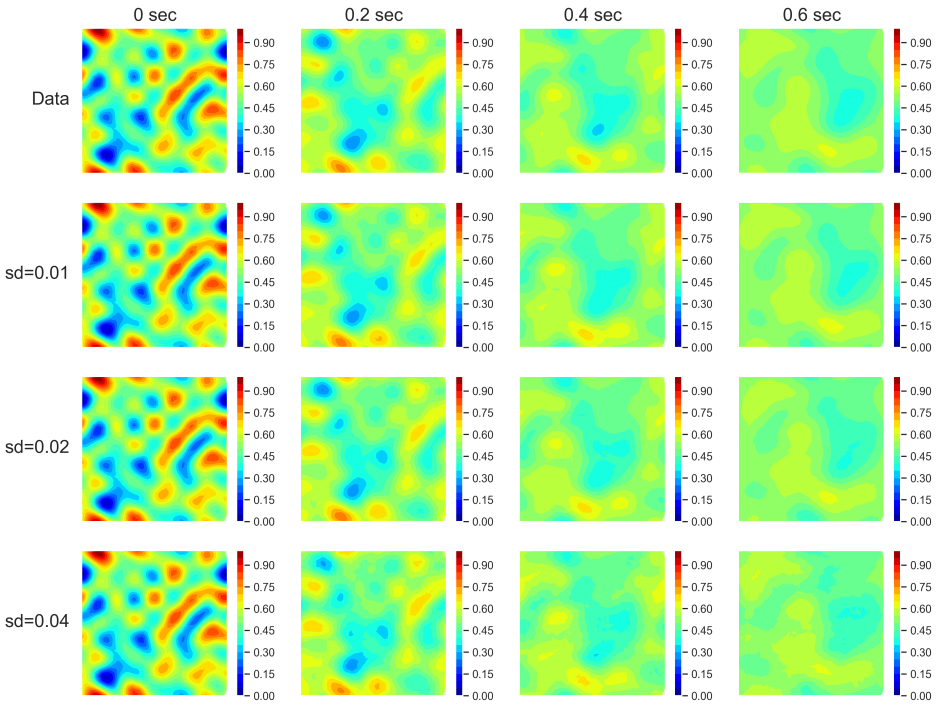

Figure 4.13. A comparison of the true states of the system with predictions of models trained on observations with different levels of noise. The first row shows the data, all following rows show predictions of models trained observations with σ set to 0.01, 0.02 and 0.04

where $N = 2$ and $\lambda_{kl}, \gamma_{kl} \sim \mathcal{N}(0, 1)$.

Simulation data used for all experiments in this section was generated with the following parameters: $\Delta t = 0.0016$ sec, $n = 5446$, $T = 0.8$ sec for training data and $T = 2.4$ sec for testing data. In all cases the number of train and test simulations was set to 24 and 50 respectively.

The model used for all experiments in this section has a single graph layer with functions $\phi^{(0)}$ and $\gamma^{(0)}$ represented by multilayer perceptrons (MLPs). Let's define a MLP with Tanh activation functions, 3 hidden layers and inputs dimension a, output dimension c and hidden dimension b by $\text{NN}(a, b, c)$. Then $\phi^{(0)} = \text{NN}(6, 60, 40)$ and $\gamma^{(0)} = \text{NN}(41, 60, 2)$ which gives approximately 20000 trainable parameters.

In order to solve the system of ODEs defined by Equation 3.5 torchdiffeq [47] Python package was used. All models were trained on a single NVIDIA Quadro P5000 GPU for 24 hours. Training hyperparameters are listed in Table 4.2.

4.4.1 Learning Coupled Nonlinear PDEs

The goal of this experiment is to show that the presented approach can be used for more complex cases than the convection-diffusion equation. Burgers' equations (4.7) is a set of two coupled PDEs with a nonlinearity.

Table 4.2. Training hyperparameters.

ODE Solver	Adaptive Grid Adams-Bashforth-Moulton
rtol/atol	1.0e-7/1.0e-7
optimizer	Rprop [48]
learning rate	1.0e-6
loss function	Mean Squared Error
batch size	24

A time grid with $\Delta t_c = 0.04$, which gives 21 time point for training, was used. No noise was added to the time points. The number of nodes in the spatial grid was set to 5004. As in the previous experiments, the relative error between the predicted and the true states of the system is used to assess the performance of the model. In this case, the model predicts a vector field. Therefore, to simplify visualizations and computation of the relative error, the vector field was converted to a scalar field by calculation the magnitude of the vector field at each node. Performance of the model evaluated on the test data is shown in Figure 4.14. A comparison of the model's predictions and the true states of the system for a random test case is shown in Figure 4.15.

Relative errors for the Burgers' equations are higher than for the convection-diffusion equation. This indicates that learning the Burgers' equations is a more difficult task. Nonetheless, the model is able to approximate the dynamics of the system reasonably accurately as shown in Figure 4.15.

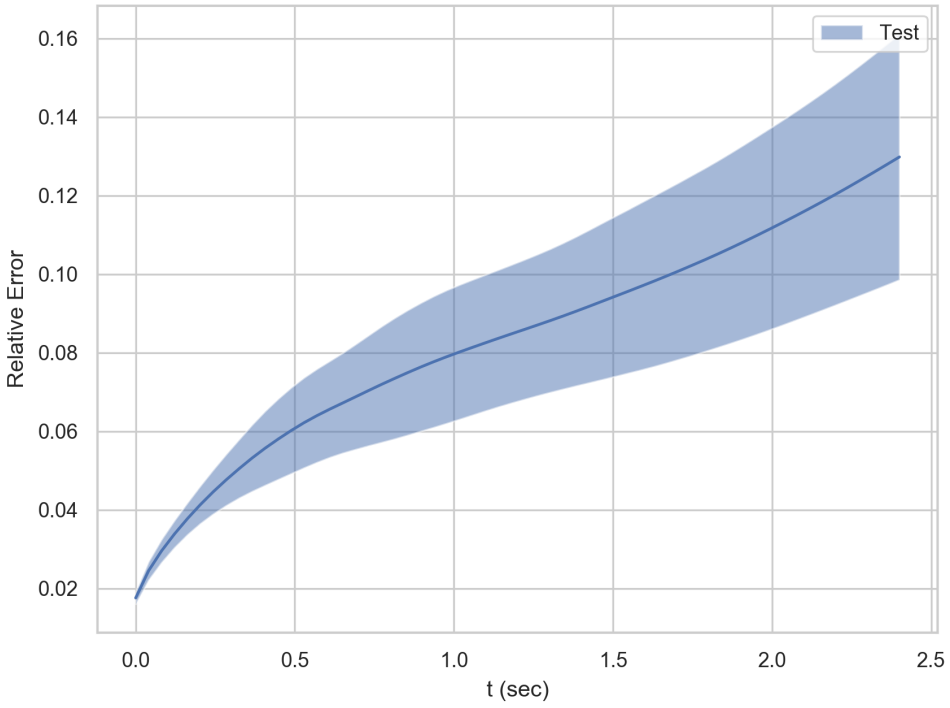

Figure 4.14. Relative test errors of the model trained on data from the Burgers' equations.

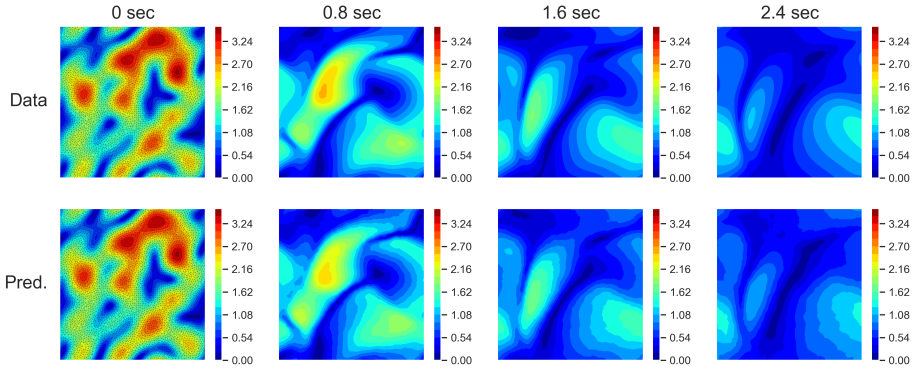

Figure 4.15. A comparison of the model's predictions and the true states of the system for the Burgers' equations. The spatial grid is shown in the left column.

4.5 Heat Equation

In this section, all solution domains are represented by a unit square with Dirichlet boundary conditions. Therefore, the following initial-boundary value problem (IBVP) is considered

$$\begin{aligned} \frac{\partial u(x,y,t)}{\partial t} &= D\nabla^2 u(x,y,t), \quad (x,y) \in \Omega,\ t \geq 0, \\ u(x,y,t) &= u_0(x,y), \quad (x,y) \in \partial\Omega,\ t \geq 0, \\ u(x,y,0) &= u_0(x,y), \quad (x,y) \in \Omega,\ t = 0. \end{aligned} \tag{4.10}$$

The diffusion coefficient D was set to 0.2. Here u is the temperature field. The initial conditions $u_0(x,y)$ were generated as follows:

$$u_0^{'}(x,y) = \sum_{k,l=-N}^{N} \lambda_{kl} \cos{(kx+ly)} + \gamma_{kl} \sin{(kx+ly)}, \tag{4.11}$$

$$u_0(x,y) = \frac{u_0^{'}(x,y) - \min u_0^{'}(x,y)}{\max u_0^{'}(x,y) - \min u_0^{'}(x,y)}, \tag{4.12}$$

where $N = 10$ and $\lambda_{kl}, \gamma_{kl} \sim \mathcal{N}(0,1)$.

Simulation data used for all experiments in this section was generated with the following parameters: $\Delta t = 0.0001$ sec, $n = 4116$, $T = 0.1$ sec for training data and $T = 0.3$ sec for testing data. In all cases the number of train and test simulations was set to 24 and 50 respectively.

The model used for all experiments in this section has a single graph layer with functions $\phi^{(0)}$ and $\gamma^{(0)}$ represented by multilayer perceptrons (MLPs). Let's define a MLP with Tanh activation functions, 3 hidden layers and inputs dimension a, output dimension c and hidden dimension b by $\text{NN}(a,b,c)$. Then $\phi^{(0)} = \text{NN}(4,60,40)$

and $\gamma^{(0)} = \text{NN}(41, 60, 1)$ which gives approximately 20000 trainable parameters.

In order to solve the system of ODEs defined by Equation 3.5 torchdiffeq [47] Python package was used. All models were trained on a single NVIDIA Quadro P5000 GPU for 24 hours. Training hyperparameters are listed in Table 4.3.

Table 4.3. Training hyperparameters.

ODE Solver	Adaptive Grid Adams-Bashforth-Moulton
rtol/atol	1.0e-7/1.0e-7
optimizer	Rprop [48]
learning rate	1.0e-6
loss function	Mean Squared Error
batch size	24

4.5.1 Generalizing to New Solution Domains

The goal of this experiment is to show that a model trained on one solution domain, for example a unit square, can be applied to a different solution domain. In this experiment a model is trained on a unit square and then tested on a unit square and on a unit square with a cutout as shown in Figure 4.16. Ideally, relative test errors should be similar for both cases.

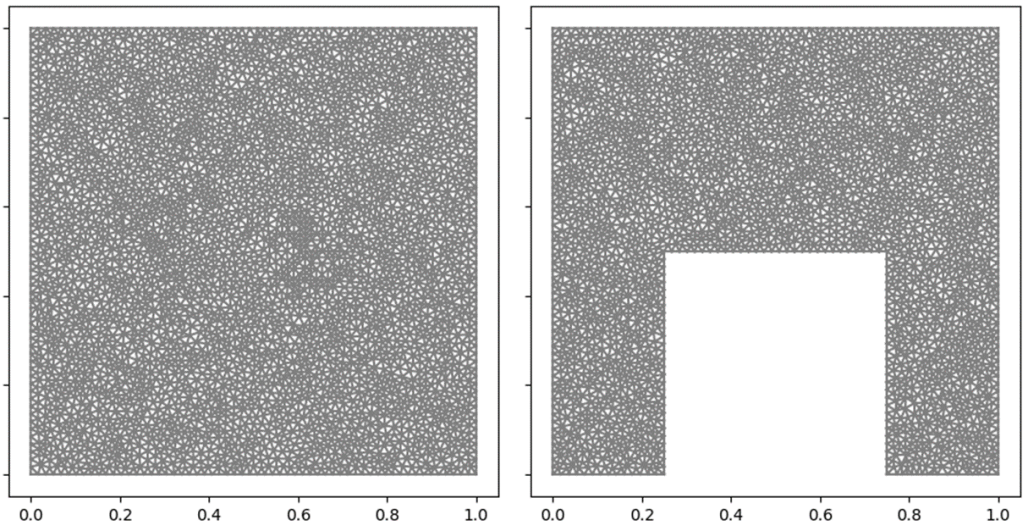

Figure 4.16. Grid shapes: unit square (left) and unit square with a cutout (right).

A time grid with $\Delta t_c = 0.005$ sec, which gives 21 time point for training, was used. No noise was added to the time points. The number of nodes in the training domain was set to 4166. Dirichlet boundary conditions were implemented by multiplying the output of the model by a 0/1 mask with zeros corresponding to nodes on the boundaries. This ensures that the state of the boundary nodes is not changing over time. As in the previous experiments, the relative error between

the predicted and the true states of the system is used to assess the performance of the model. Performance of the model evaluated on the test data with different grid shapes is shown in Figure 4.17. A comparison of the model's predictions on the unit square with a cutout and the true states of the system for a random test case is shown in Figure 4.18.

As can be seen, the model's performance on both domains is comparable. This shows that the model can be used on new grids without significant decrease in performance.

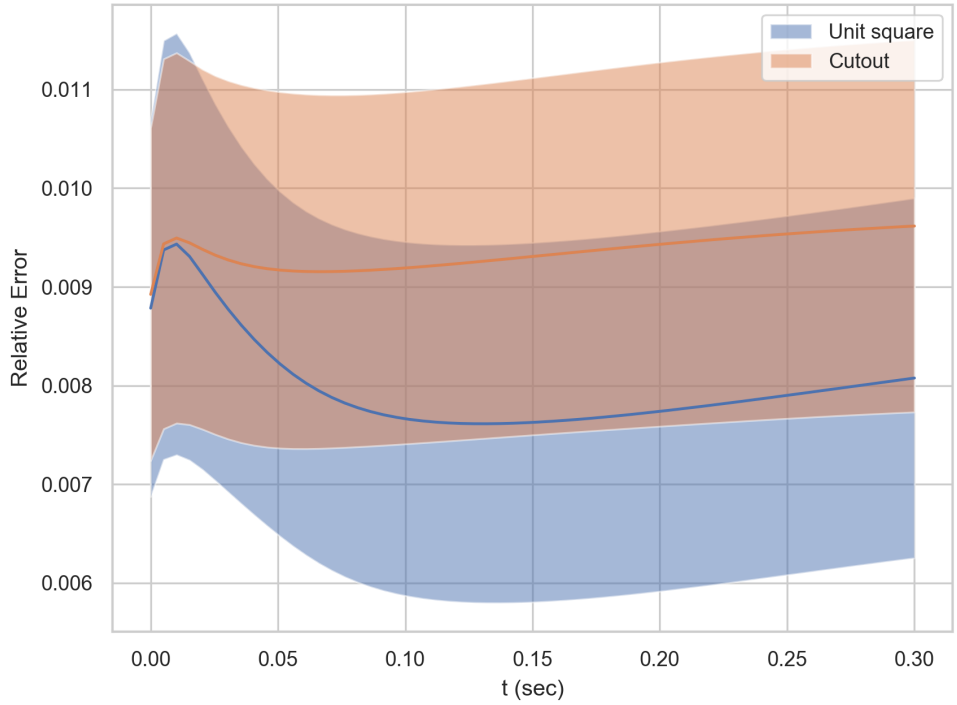

Figure 4.17. Relative test errors for different grid shapes.

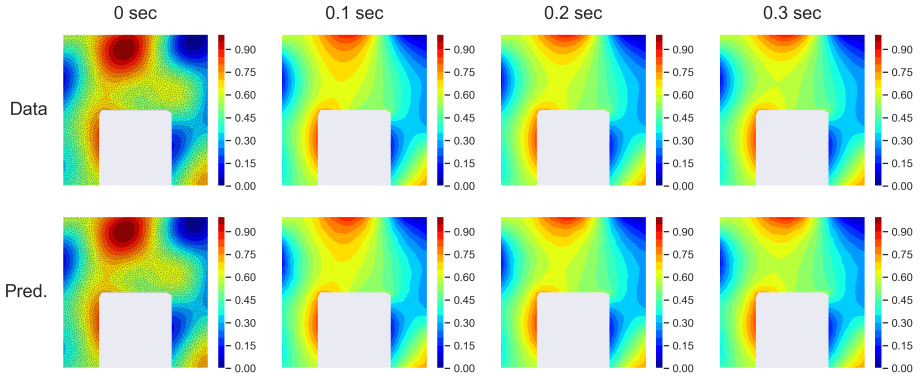

Figure 4.18. A comparison of the model's predictions on the unit square with a cutout and the true states of the system for the heat equation. The spatial grid is shown in the left column.

5. Discussion

As was stated in Chapter 1, the goal of this work was to develop a continuous-time graph-network-based surrogate model for time-dependent PDEs. Experiments conducted in Chapter 4 show that this goal was achieved.

The model was trained on data obtained by solving three different PDEs: the convection-diffusion equation, the Burgers equations, and the heat equation. The first set of experiments, presented in Section 4.3, was conducted with data obtained by solving the convection-diffusion equation. The goal of these experiments was to show how the model behaves on spatial and time grids of different sizes, how irregular time steps between observations affect the model's performance and how robust the model is to noise in the observations. The second set of experiments, presented in Section 4.4, showed that the model is able to learn systems of coupled nonlinear PDEs with reasonable accuracy. Finally, experiments presented in Section 4.5 show that after training the model on a simple grid (e.g. a unit square) it can generalize to grids with various shapes without a significant decrease in prediction performance.

In Section 4.3 the model was trained on data obtained by solving the convection-diffusion equation with a constant diffusion coefficient and a uniform velocity field. Results presented in Subsection 4.3.1 show how the model performs on grids ranging from fine to coarse (Figure 4.4). As can be seen from Figure 4.5, the performance of the model improves with the number of nodes in the grid. As shown in Figure 4.6, for fine grids with 2991 and 2244 nodes the model shows good predictive accuracy and for the coarsest grid with 749 nodes the model is still able to capture large-scale features of the solution reasonably well.

The experiment conducted in Subsection 4.3.2 shows how the timestep between observations in the training data affects the performance of the model. Figures 4.7 and 4.9 demonstrate that the model is able to achieve good predictive performance even when trained on only four time points but fails when only the initial and final time points are given. Therefore, there exists a maximum allowable step size

which can be used for training the model. An example of the training data with four time points is shown in Figure 4.9.

Results in Subsection 4.3.3 show that the model trained on an irregular time grid is able to achieve the same performance as the model trained on a time grid with a constant time step (Figure 4.10). The model trained on the irregular time grid performs slightly better either due to a better initialization or due to a better placement of the time points.

In subsection 4.3.4 the model was trained on data with different levels of noise. Figures 4.12 and 4.13 show that the even for the highest level of noise, the model is still able to learn a reasonably accurate approximation of the dynamics of the system. This demonstrates that the model is robust to noise in the training data.

In all previous experiments, the model was trained on data obtained by solving the convection-diffusion equation which is a linear scalar PDE. In Section 4.4 the model was trained on data obtained by solving the Burgers' equations which is a set of two coupled nonlinear PDEs. As demonstrated in Subsection 4.4.1 and Figures 4.14 and 4.15 the model is able to learn the Burgers' equations and achieve reasonable predictive performance. Differences between the data and predictions of the model are larger than for the convection-diffusion equation. This indicates that learning the Burgers' equations is a more difficult task than learning the convection-diffusion equation. Better results probably could be achieved by using longer training times, more data, different hyperparameters or a different model architecture.

Finally, in Section 4.5 the model's capability to generalize to new solution domains is tested. The model is trained on data obtained by solving the heat equation on a unit square. The testing data was obtained by solving the same equation on grids of two different shapes: a unit square and a unit square with a cutout (Figure 4.16). Figures 4.17 and 4.18 show that the model can be applied to new solution domains without a significant decrease in the predictive performance. This behavior is expected since the model is based on MPNNs which do not depend on the shape of the graph.

The experimental results suggest that the approach to learning partial differential equations presented in this work possesses all the properties stated in Chapter 1 and can be used as a surrogate model for dynamical PDE-driven systems with known or unknown governing equations.

6. Conclusion

There are systems whose governing equations are either unknown or only partially known. The state of such systems can be observed at some spatial points over time. The collected data can be used to learn a model that approximates the underlying governing equations of the observed system. The observation points might be randomly located and the time intervals between observations might not be constant. Previously developed models had difficulties working with such data. This work proposed a model which is able to work with unstructured grids and random time intervals between observations.

There are two main components of the models. First, it is based on message-passing neural networks which allows it to work with randomly located observation points. Second, it learns the temporal derivative of the state of the system and uses ODE solvers to evolve the state forward in time. It allows training the model on data with random time intervals. The model was shown to work well on spatial grids with different numbers of nodes and time grids with different and random time intervals. Also, the model is able to generalize to grids of different shapes and can learn complex PDEs such as the Burgers' equations.

Despite the apparent success of the presented model, there are many directions for improvements. The model's accuracy might be insufficient. Sometimes, differences between the true states of the system and the model's predictions can be seen even by the naked eye. This problem could be solved by providing more data, selecting better hyperparameters, training for longer or using a different model architecture. It is not clear which of these approaches will yield the largest improvement. Furthermore, training the model is slow. The main reason for that is the need to solve a system of ODEs during both the forward and backward passes. At this point, the only solution to this problem seems to be a more efficient implementation of ODE solvers. Also, training of the model might be unstable. This is typically manifested through random increases in the value of the loss function during training. This problem usually can be solved by using a smaller

learning rate, tighter ODE solver tolerances or smaller time intervals between observations in the training data.

Bibliography

[1] Mario Ohlberger and Stephan Rave. Reduced basis methods: Success, limitations and future challenges. *Proceedings of the Conference Algoritmy*, pages 1–12, 2016.

[2] B. A. Mckinney, J. E. Crowe, H. U. Voss, P. S. Crooke, N. Barney, and J. H. Moore. Hybrid grammar-based approach to nonlinear dynamical system identification from biological time series. *Physical Review E*, 73(2), 2006.

[3] J. Bongard and H. Lipson. Automated reverse engineering of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 104(24):9943–9948, Jun 2007.

[4] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, Mar 2009.

[5] John R. Koza. *Genetic Programming On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[6] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[7] Samuel H. Rudy, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3(4), 2017.

[8] Hao Xu, Haibin Chang, and Dongxiao Zhang. Dl-pde: Deep-learning based data-driven discovery of partial differential equations from discrete and noisy data. *arXiv preprint arXiv:1908.04463*, 2019.

[9] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[10] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[11] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.

[12] Mohammad Amin Nabian and Hadi Meidani. A deep neural network surrogate for high-dimensional random partial differential equations. *arXiv preprint arXiv:1806.02957*, 2018.

[13] Alexandre M Tartakovsky, Carlos Ortiz Marrero, Paris Perdikaris, Guzel D Tartakovsky, and David Barajas-Solano. Learning parameters and constitutive relationships with physics informed deep neural networks. *arXiv preprint arXiv:1808.03398*, 2018.

[14] Yinhao Zhu, Nicholas Zabaras, Phaedon-Stelios Koutsourelakis, and Paris Perdikaris. Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. *Journal of Computational Physics*, 394:56–81, 2019.

[15] Qianxiao Li, Felix Dietrich, Erik M Bollt, and Ioannis G Kevrekidis. Extended dynamic mode decomposition with dictionary learning: A data-driven adaptive spectral decomposition of the koopman operator. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(10):103111, 2017.

[16] Matthew O Williams, Ioannis G Kevrekidis, and Clarence W Rowley. A data–driven approximation of the koopman operator: Extending dynamic mode decomposition. *Journal of Nonlinear Science*, 25(6):1307–1346, 2015.

[17] Xuping Xie, Clayton Webster, and Traian Iliescu. Closure learning for nonlinear model reduction using deep residual neural network. *Fluids*, 5(1):39, 2020.

[18] Jonathan B Freund, Jonathan F MacArt, and Justin Sirignano. Dpm: A deep learning pde augmentation method (with application to large-eddy simulation). *arXiv preprint arXiv:1911.09145*, 2019.

[19] Rohit K Tripathy and Ilias Bilionis. Deep uq: Learning deep neural network surrogate models for high dimensional uncertainty quantification. *Journal of computational physics*, 375:565–588, 2018.

[20] Yuehaw Khoo, Jianfeng Lu, and Lexing Ying. Solving parametric pde problems with artificial neural networks. *arXiv preprint arXiv:1707.03351*, 2017.

[21] Yinhao Zhu and Nicholas Zabaras. Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification. *Journal of Computational Physics*, 366:415–447, 2018.

[22] Nicholas Geneva and Nicholas Zabaras. Modeling the dynamics of pde systems with physics-constrained deep auto-regressive networks. *Journal of Computational Physics*, 403:109056, 2020.

[23] Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. Pde-net: Learning pdes from data. *arXiv preprint arXiv:1710.09668*, 2017.

[24] Shaowu Pan and Karthik Duraisamy. Long-time predictive modeling of nonlinear dynamical systems using neural networks. *Complexity*, 2018, 2018.

[25] Gavin D Portwood, Peetak P Mitra, Mateus Dias Ribeiro, Tan Minh Nguyen, Balasubramanya T Nadiga, Juan A Saenz, Michael Chertkov, Animesh Garg, Anima Anandkumar, Andreas Dengel, et al. Turbulence forecasting via neural ode. *arXiv preprint arXiv:1911.05180*, 2019.

[26] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in neural information processing systems*, pages 4502–4510, 2016.

[27] Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. A compositional object-based approach to learning physical dynamics. *arXiv preprint arXiv:1612.00341*, 2016.

[28] Katerina Fragkiadaki, Pulkit Agrawal, Sergey Levine, and Jitendra Malik. Learning visual predictive models of physics for playing billiards. *arXiv preprint arXiv:1511.07404*, 2015.

[29] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[30] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. *arXiv preprint arXiv:1806.01242*, 2018.

[31] Sungyong Seo and Yan Liu. Differentiable physics-informed graph networks. *arXiv preprint arXiv:1902.02950*, 2019.

[32] Ferran Alet, Adarsh K Jeewajee, Maria Bauza, Alberto Rodriguez, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Graph element networks: adaptive, structured computation and memory. *arXiv preprint arXiv:1904.09019*, 2019.

[33] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[34] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[35] Andrew R Barron. Approximation and estimation bounds for artificial neural networks. *Machine learning*, 14(1):115–133, 1994.

[36] Kurt Hornik, Maxwell Stinchcombe, Halbert White, et al. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[37] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

[38] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[39] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[40] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

[41] Mark S. Gockenbach. *Understanding And Implementing the Finite Element Method*. Society for Industrial and Applied Mathematics, USA, 2006.

[42] S. Larsson, V. Thomee, and Springer Berlino. *Partial Differential Equations with Numerical Methods*. Texts in Applied Mathematics. Springer, 2003.

[43] Lev Semenovich Pontryagin. *Mathematical theory of optimal processes*. Routledge, 2018.

[44] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[45] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[46] William E Schiesser. *The numerical method of lines: integration of partial differential equations*. Elsevier, 2012.

[47] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *Advances in Neural Information Processing Systems*, 2018.

[48] Martin Riedmiller and Heinrich Braun. Rprop - a fast adaptive learning algorithm. Technical report, Proc. of ISCIS VII), Universitat, 1992.